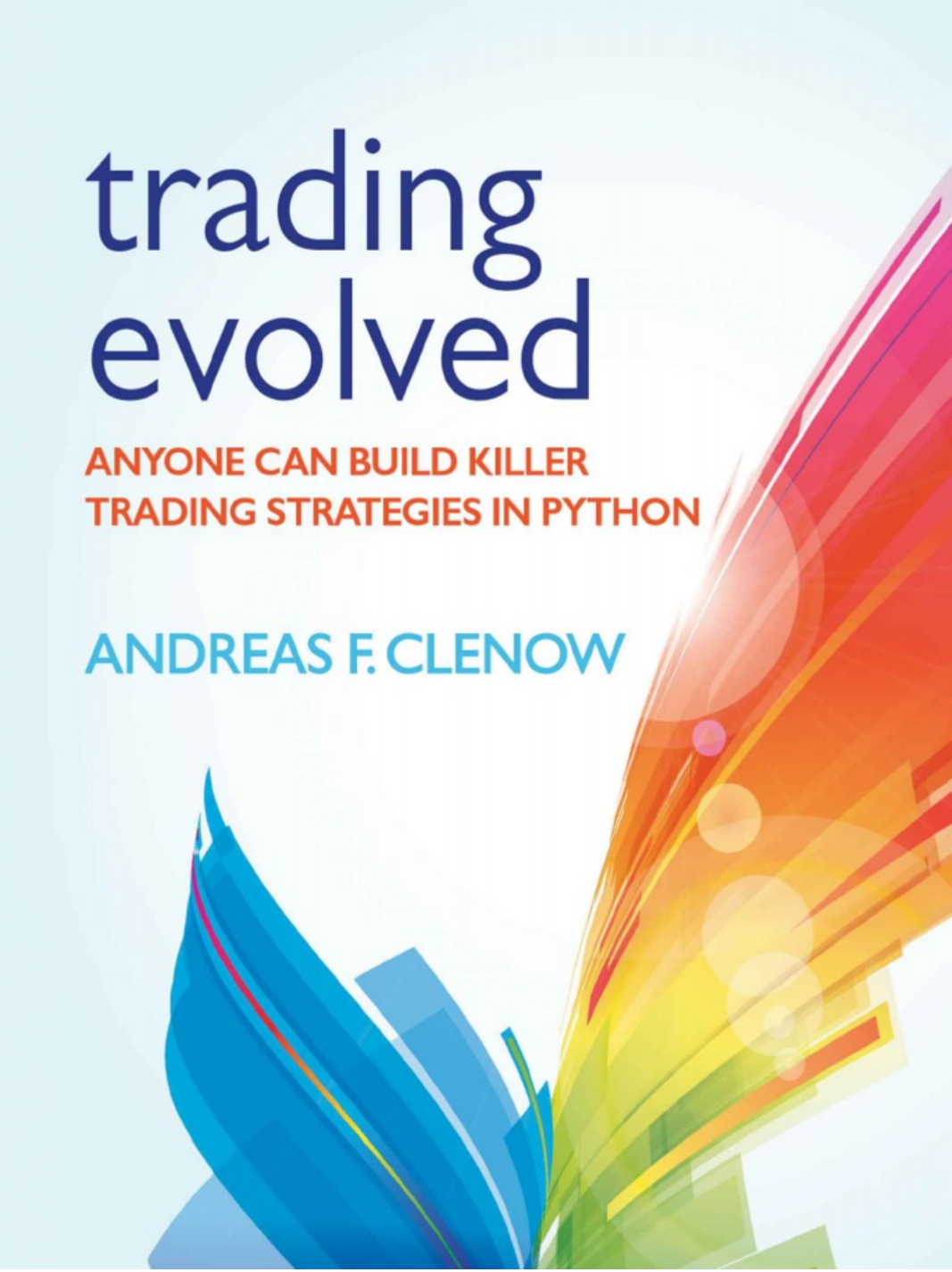
trading evolved

ANYONE CAN BUILD KILLER TRADING STRATEGIES IN PYTHON

ANDREAS F. CLENOW



VISIT...

Trading Evolved

Anyone can Build Killer Trading Strategies

in Python

Book version 1.1

Copyright © 2019 Andreas F. Clenow
Registered Office: Equilateral Capital Management GmbH, Lowenstrasse 1, 8001 Zurich, Switzerland

For details of editorial policies and information for how to apply for permission to reuse the copyright material in this book please see our website at www.FollowingTheTrend.com.

The right of the author to be identified as

the author of this work has been asserted in accordance with the Copyright, Designs and Patents act 1998. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by any means, electronic, mechanical, photocopying, teleportation, recording or otherwise except as permitted by the UK Copyright, Designs and Patents Act 1998 or other irrelevant but good sounding legal paragraphs, without the prior permission of the publisher. Doing so would make you a very naughty boy and you don't want that, do you.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book except if expressly stated.

Limit of Liability/Disclaimer of Warranty: While the publisher and the author have used their best efforts in preparing this book, they make no representations or warranties with the respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

ISBN: 9781091983786

To my wife Eng Cheng and my son Brandon, not only because of their love and support but also because they have been asking for months about whether they will get a dedication in this book.

1 About this Book

The Trading Strategies in this Book
How to Read this Book
How this book is written
How the code is written
Errata
Support

2 Systematic Trading

Trading Approach Validation
Scientific Approach
Consistent Methodology
Time Management

3 Developing Trading Models

Model Purpose
Rules and Variations
Handling Data
Asset Class

Investment Universe
Allocation and Risk Level
Entry and Exit Rules
Rebalancing

4 Financial Risk
Quantifying Risk
Mark to Market
Common Risk Fallacies
Risk as Currency to Buy Performance

5 Introduction to Python
Some Assembly Required
Python Emerges as the Logical Choice
Programming Teaching Approach
Installing Python on your Computer
Let’s Run Some Code
Working with Jupyter Notebook
Dictionary Lookup
Conditional Logic
Common Mistakes
Installing Libraries

6 Bring out the Pandas
Documentation and Help
Simple Python Simulation
Making a Correlation Graph
Prettier Graphs

7 Backtesting Trading Strategies

Python Backtesting Engines
Zipline and Quantopian
Pros and Cons
Installing Zipline
Problems with Installing Zipline
Patching the Framework
Zipline and Data
Ingesting the Quandl Bundle
Installing Useful Libraries
Where to Write Backtest Algos
Your First Zipline Backtest
Portfolio Backtest
Data Used for this Book

8 Analyzing Backtest Results

Installing PyFolio
Portfolio Algorithm to Analyze
Analyzing Performance with PyFolio
Custom Analysis
Day Snapshot
Custom Time Series Analytics

9 Exchange Traded Funds

The Good
The Bad
The Worst
Shorting Exchange Traded Funds

10 Constructing ETF Models
Asset Allocation Model

11 Equities
The Most Difficult Asset Class
A word on Methodology
Equity Investment Universe
Dividends

12 Systematic Momentum
Replicating this Model
Momentum Model Rules Summary
Investment Universe
Momentum Ranking
Position Allocation
Momentum Model Logic
Downside Protection
Momentum Model Source Code
Performance
Equity Momentum Model Results

13 Futures Models
Futures Basics
Futures Mechanics and Terminology
Futures and Currency Exposure
Futures and Leverage

14 Futures Modeling and

Backtesting
Continuations
Zipline Continuation Behavior
Contracts, Continuations and Rolling

15 Futures Trend Following
Principles of Trend Following
Revisiting the Core Trend Model
Model Purpose
Investment Universe
Trading Frequency
Position Allocation
Entry Rules
Exit Rules
Costs and Slippage
Interest on Liquidity
Trend Model Source Code
Core Trend Model Results

16 Time Return Trend Model
Investment Universe
Trading Frequency
Position Allocation
Trading Rules
Dynamic Performance Chart
Time Return Source Code
Time Return Model Performance
Rebalancing

17 Counter Trend Trading

Counter Model Logic
Quantifying Pullbacks
Rules Summary
Counter Trend Source Code
Counter Trend Results

18 Trading the Curve

Term Structure Basics
Quantifying Term Structure Effect
Curve Model Logic
Curve Trading Source Code
Curve Trading Results
Model Considerations

19 Comparing and Combining Models

Combining the Models
Implementing a Portfolio of Models

20 Performance Visualization and Combinations

Storing Model Results
How the Model Performance Analysis was done
How the Combined Portfolio Analysis was done

21 You can't beat all of the Monkeys all of the Time
Mr. Bubbles goes to Wall Street
The Problem is with the Index
Finding Mr. Bubbles

22 Guest Chapter: Measuring Relative Performance

23 Importing your Data
Making a Bundle
Zipline and Futures Data
Futures Data Bundle
Patching the Framework

24 Data and Databases
Your Very Own Securities Database
Installing MySQL Server
Making an Equities Time-Series Table
Populating the Database
Querying the Database
Making a Database Bundle

25 Final Words – Path Forward
Build Your Own Models
Other Backtesting Engines
How to Make Money in the Markets

References

Index

While I would like to claim that I managed to write this book with absolutely no help from anyone, nothing could be farther from the truth. The help that I've received along the way has been invaluable, and I couldn't have completed this book without it. Without these people, this book would either never have gotten off the ground or it would have ended up a total disaster. In no particular order, I'd like to express my gratitude towards these people.

John Grover, Matthew Martelli, Robert Carver, Riccardo Ronco, Thomas Starke, Tomasz Mierzejewski, Erk Subasi and Jonathan Larkin.

1 About this Book

This book will guide you step by step on how to get familiar with Python, how to set up a local quant modeling environment and how to construct and analyze trading strategies. It's by no means an exhaustive book, either on Python, backtesting or trading. It won't make you an expert in either topic, but will aim for giving you a solid foundation in all of them.

When approaching something as complex as backtesting trading strategies, every step on the way can be done in a multitude of ways. This book does not attempt to cover all the different ways

that you could approach Python trading strategies. A book like that would require many times the text mass of this book. But more importantly, a book of that type would likely scare away the bulk of the people that I want to address.

The point of my books, all of my books, is to make a seemingly complex subject accessible. I want to take a subject matter which most people find daunting and explain it in a way that a newcomer to the field can understand and absorb.

My first book, Following the Trend (Clenow, Following the Trend, 2013), was based on exactly that premise. Having spent some time in the trend following hedge fund world, it surprised me when I realized how many myths and misunderstandings surrounded this particular type of trading strategy. When I decided to write an entire book just to explain one fairly simple trading strategy, I had not expected the incredible reception that it received worldwide. It was great fun to start off with an international best seller, and no one was more surprised than I was over the

publicity that this book received.

My second book, Stocks on the Move (Clenow, Stocks on the Move, 2015), was the result of a very common question that I kept getting. “Can the trend following approach be applied to stocks?” My initial, instinctive reply to that questions was “sure, but you would need to modify the rules a bit”. After giving it some thought, I realized that the topic warrants a book by itself, and that equity momentum models differ enough from trend following to be classified as a different strategy. Out of this, Stocks on the Move was born.

In my first two books, I tried hard to make everything accessible. Not just understandable, but explained in such detail that anyone reading it would be able to replicate it. I researched low cost software and data, making sure that it’s within anyone’s budget and that the trading strategies and backtests could be reconstructed by readers so that any claim I made could be verified.

It was great to see all the mails over

the years from readers who went through the process to replicate and test it all. But there were also many mails from readers who lacked the technical skills to construct the backtests. Many used backtesting software which is far too simplistic for any serious portfolio modeling, and some wouldn't know where to start with setting up a more robust environment.

Out of this, a new book idea was born. A book focused on making quantitative backtesting of trading strategies accessible to anyone.

The Trading Strategies in this Book

This is not a book full of super-secret trading strategies which will turn a thousand dollars into a million next week. While I like to think that there are some clever trading strategies in this book, it's

not meant to be cutting edge, revolutionary stuff. I would think that most readers will learn some interesting things about trading models, but that's not the main point of this book.

In order to teach you about using Python to test trading ideas, I need to show some trading ideas to test. I will show you a few models which I hope will be helpful. You will see complete trading models for ETFs, stocks and futures of varying degree of complexity. I will use these trading strategies as tools to explain what capabilities you will need and how to go about making these strategies real.

I often stress in my books that any trading strategy shown is a teaching tool and not meant for you to go out and trade. I will repeat this statement a few times throughout the book. I would strongly discourage anyone from copying anyone else's trading strategies and blindly trading them. But this is after all where this book comes in.

What I do recommend is that you read about other people's trading

strategies. Learn from them. Construct a suitable backtesting environment and model the strategies. Then figure out what you like and what you don't like. Modify the parts you like, find ways to incorporate them into your own approach and come up with ways to improve the way you trade.

You need to make the models your own to fully understand them, and you need to understand them to fully trust them. This book will give you the necessary tools and skillset to get this done.

How to Read this Book

If your mind is set on taking quantitative backtesting of trading strategies seriously, then this is a book that you will probably want to spend quite a bit of time with. For many books, the contents can easily be digested by reading them once, cover to cover. That's

probably the case with my previous books, which each had a fairly small amount of key information to pass on, spending a lot of pages doing it.

As opposed to my previous books, this one has quite a bit of source code in it. My guiding principle has been that anyone with a fair understanding of computers and trading should be able to understand and absorb the contents of this book, without any prior programming knowledge required.

But I have also assumed that anyone who does not have experience building programming code will likely need to go over the book multiple times, with a computer nearby to try the code samples.

This is a practical book, and there is no substitute to actually trying things out for yourself.

My suggestion is that you start by reading through the entire book once, cover to cover. That gives you a good overview of the subject matter and you will find which areas are of most interest

to you.

If Python is new to you, start off with the easier examples early in the book. Make sure that you understand the basics before moving to the heavy stuff.

The most difficult part in this book, from a technical point of view, is probably to get your own data imported into the Zipline backtester so that you can run backtests based on that. I have tried to make this part as easy as I can for you, but there are plenty of ways that things can go wrong.

My most important advice here is not to give up. The initial learning curve may be a little steep, but few things worthwhile are easy. Don't get scared off by the tech vocabulary and the scary looking code. Yes, this will take a bit of work for most people to learn, but really, anyone can learn this.

How this book is written

My intention is to give you a step by step guide, starting with the assumption of zero technical knowledge, and ending with you having the skillset to construct complex quantitative trading strategies.

This is a practical guide and you may want to have a computer nearby. My assumption has been that most readers will probably read the book, or a chapter, and then go back to the computer to try out what they have read about, with the book next to them.

The book is very much written in order, at least until you start reaching the more advanced chapters. Each chapter assumes that you have acquired the knowledge from the previous. This means that unless you are already proficient in Python, you probably want to read the book in order.

How the code is written

Programming code, just like books, can be written in many different styles and for many different purposes. The code in this book is written to be clear and easy to understand. It may not always be the most efficient way of writing the code, or the fastest to execute. But that's not the point here. The point is for you to learn.

Often there are tasks described which could be done in any number of different ways. As much as possible, I have tried to simply pick one approach and stick with it, rather than describing all the other possible ways it could have been done.

There are other great books if you just want in-depth knowledge of Python, such as Python for Data Analysis (McKinney, 2017) or Python for Finance (Hilpisch, 2018). And if your interest is in a deep dive into systematic trading, you should look at something like the aptly named Systematic Trading (Carver, Systematic Trading, 2015).

This book is an intersection of these ideas. It's not as deep on either topic, but

in return it gives you the practical skills to combine the topics and achieve a tangible result.

This book contains quite a bit of source code. All programming code is highlighted using a `different font and backgroun d` . You will see this throughout the book, to differentiate descriptive text from code. In the same way, names of libraries and key technical terms are shown in **bold text** .

Errata

A book on this type of topic is sure to contain errors. While I have done my best to avoid it and had multiple people review the code, experience tells me that there are still errors and mistakes. This is the nature of programming code.

Hopefully there are no major errors, but there are sure to be some. When errors are found and confirmed, I will post updates on the book website, **www.followingthetrend.com/trading-evolved** . If there is enough requests for

it, I may set up a forum there as well for readers to discuss their experiences of backtesting strategies with Python.

Support

I try to be an accessible guy and I have greatly enjoyed the large amount of email since my first book was published. I got to know a great many interesting individuals from all over the world and I stay in contact with many of them.

One thing that worried me when I set out to write this book though, was the type of mails it's likely to result in. The kind asking for technical advice, help with code, debugging and such. I'm afraid to say that I don't have the time to help out on these matters. If I did, I would simply be stuck full time debugging other people's code.

Please understand if I am unable to help you out with technical issues. I do enjoy communicating with all you readers out there, but I cannot offer individual support. I hope that you understand and I

will do my best to include everything in this book and to keep the book website updated with errata and related articles.

2 Systematic Trading

This book is about systematic trading. That is, the use of computers to model, test and implement mathematical rules for how to trade. This way of working is by no means a shortcut, as it may unfortunately appear to some newcomers in the field. Building solid mathematical models with predictive value going forward can be a daunting task. It takes hard work and research. But on the other hand, this way of working allows you to test what you believe about the market, and even more importantly to test how your ideas about trading would have performed in the past.

Trading Approach

Validation

A good reason to enter the space of systematic trading and quantitative modelling is to validate your ideas, or others' ideas for that matter. You have probably read books or websites telling you to buy when some indicator hits some value, on lines crossing or something similar. Perhaps that's a valid approach, and perhaps not. You could find out by taking a bet on these rules, or you could formulate them into a trading model and test them.

When you first starting testing ideas in this way, it can be an eye opener. The first thing you will likely realize is that most such advice represent only a small part of the puzzle, or that they can be difficult or even impossible to turn into firm trading rules.

Whether or not you want to go the full mile and implement completely systematic trading rules, the ability to test and validate ideas is very valuable. Many market participants rely on old platitudes

and so called common knowledge about how the markets work, without ever bothering to validate these ideas.

Take something as common as the saying “Sell in May and go away”, which refers to the common belief that markets perform best during the winter. It does not take any deeper understanding of systematic trading to find out if there is a value in such an approach.

Other wisdoms you may have heard would include how you should hold 70% stocks and 30% bonds in the long run, how you should never hold stocks in October or similar. After reading this book, you should have a sufficient toolkit at your disposal to try out any such ideas you may hear.

Having this ability to test ideas also tends to aid critical thinking. Once you understand what is required to test an idea, the logical parts that make up a complete trading model and the details that you need to sort out, you will quickly see if a proposed method of trading is possible to model or not. This will help

you understand if what is being suggested is a complete method, or just a small part of it.

Once you start thinking in this manner and look for ways to convert someone's text into trading rules, you will start to apply more critical thinking to claims surrounding various approaches to the markets. Many seemingly mathematical approaches to the market are not in any way quantifiable, once you start trying to construct a model around them. Try to implement Fibonacci or Elliot Wave ideas into a testable approach, and you will find yourself in a bottomless logical pit with a china teapot floating in the middle.

Scientific Approach

Systematic trading aims for a scientific approach. I would say aims for, as a most systematic traders take shortcuts which an academic researcher would take issue with. As practitioners, systematic traders are not in the business of seeking

truth, but rather in the business of making money. At times, this can mean that some scientific principles may be cut short, but it's important to keep the principles intact.

In science, hypotheses are formulated and tests for them are devised. The default assumption is always that any given hypothesis you may have is false, and if the tests attempt to demonstrate this. If the tests fail to show validity of our hypothesis, it's rejected.

This is the key difference between gut feeling trading and a somewhat scientific approach. The willingness to throw away your ideas and reevaluate your beliefs if they can't demonstrate real world results.

In order to do this, you first need for formulate every aspect of your hypothesis into firm rules. This in itself is a valuable skill to have, to be able to break down your ideas into logical components.

The next step is to construct a test for these rules. For that, you need a backtesting environment capable of

testing your rules. You also need relevant data, and to make sure that this data is correct, clean and properly suitable for testing your ideas. Depending on your choice of asset class, time frame and complication, this might be relatively simple and cheap, or it may get a little tricky.

When constructing and executing your tests, the so called backtests, you should always have a skeptical mindset. Your default way of thinking should be to find ways to reject the rules. To show that they fail to add value and should be discarded. That you need to start over.

The other way of working, to find ways to show the value of your rules, is easy. If you purposely try to construct tests to show how clever your rules are, your own confirmation bias will push you to accept ideas which are unlikely to have predictive value going forward.

Backtesting is the process of applying a set of trading rules on historical price series, to investigate what would theoretically have happened if you

had traded them in the past. This book will go into details on how you can use Python to set up such a backtesting environment and how to write code to test historical performance.

This book will not however go into any level of depth on applying scientific principles to the various aspects of constructing trading models. That's a vast subject which would require a book all by itself. Luckily, a good such book already exists (Carver, Systematic Trading, 2015).

Consistent Methodology

Trading can often be emotionally exhausting. Discretionary trading requires a constant focus and can be greatly dependent on your mental and emotional state on any given day. External factors can easily affect your trading performance. If you are having relationship issues, if a loved one has a health situation or even if your favorite

football team just lost an important game, you may find that your temper or lack of focus can greatly impact your performance.

It can also be factors directly related to your trading that clouds your mind. If you just took a big loss for instance, you may find yourself trying to make the market give you your money back, or to prove yourself and your abilities by trading more aggressively. Depending on your personality, a loss might also make you gun shy and have you trade more defensively, or not at all.

During times of market distress, this phenomenon impacts most of us. When there are big headlines on the news ticker, making prices crash and showing wild intraday swings, most people lose money. And most people spend their days being upset, scared or with high adrenaline. That's not a great state of mind for most people to make important decisions.

While some personality types thrive in the high pressure environments that market crises cause, most people make

very poor decisions in such situations.

This is where systematic trading really shines. It will remove this emotional aspect of trading, by providing clear cut rules. If you have done your job well and constructed solid trading rules, you simply let them do their thing.

When the market crashes and everyone around you is in panic, you can calmly continue to follow the rules, in the knowledge that they have been tested for this type of market climate and you know what to expect. There is no need to make rash decisions under fire. Just follow your rules.

Even during more normal market circumstances, you can achieve a more consistent and more predictable performance with a rule bound, systematic approach. Whether you can achieve higher returns or not is of course a wholly unrelated question.

Time Management

Most systematic traders don't have

a need to sit by the screens and watch the markets all day. That does not necessarily mean that they can spend all day at the beach, but they are generally freer to plan their day.

Staring at tick charts all day can be quite addictive. If you trade on a shorter time frame and make your decisions based on what you see in the market, you probably do need to sit in front of your Bloomberg or Reuters all day. Given the global market trading and exchange opening hours, this might mean that you will never really be off duty.

Many systematic traders work hard and long hours, but you do have a much greater degree of flexibility. If your purpose is to trade your own personal account, you could develop rules that trade daily, weekly or even monthly. This would allow you to trade as a hobby and keep your regular day job. You can develop rules that fit your schedule.

Most systematic traders execute their trades manually, in particular in the hobby segment. That is, even if the rules

are exact and all trading signals are followed, the task of entering the trades is still yours. There is really nothing wrong with working in this manner, as long as you can keep yourself from overriding and trying to outsmart your tested rules.

You might for instance have a trading model which trades at the opening of the exchange every day. Your trading model generates daily lists of trades for the day, and you enter them in the market before work each day. That's a common approach to longer term trading models for hobby traders.

As you get more advanced, you might even automate your trades, and have your code send the orders straight to the broker. This can be very convenient and allow you to trade faster, but it also comes with the added danger of bugs in your code. A decimal point wrong, and you end up with ten times exposure, which can make or break your day real fast. Don't go the automated route before you really know what you are doing.

An important point to understand in

this context is that even if your model is automated, it should never be unsupervised. It's a seductive idea to just train an algo to trade for you and go on vacation, and then to return to find a few more millions on your account.

Leaving a trading model to its own devices, letting it trade unsupervised without someone constantly watching is not a great idea. Computers are only as smart as the person programming it, and usually not even that smart. Constantly monitor automatic trading models.

3 Developing Trading Models

Trading strategies can be broken down to a set of components. These components are always part of a trading strategy, or at least they should be. Failure to pay attention to all of them is likely to result in a flawed and non-performing model.

Too often, people pay far too much attention to just one of these components, and glossing over the rest. The one that seems to get the most attention is the entry method. How to decide when to open a position.

The fact of the matter is that the importance of entry method varies greatly. For some types of trading strategies, the entry method is critical. For other methods, it does not matter all that much. For a long term trend following model for instance, the exact entry method and timing is not very important. For a short term mean reversal model, the entry approach is critical.

Model Purpose

Yes, your model needs to have a purpose. And no, that purpose is not " *to make money* ". Any trading model worth its salt is designed for a specific purpose, trading a specific market phenomenon to achieve a specific goal. If you don't know what your model purpose is, odds are that

all you have got is a bunch of indicators thrown together and varied until a simulation showed some positive returns. A set of over optimized rules, which are very likely to fail in reality. A solid model trades a real market phenomenon, aiming for a certain type of return profile.

What you really want to avoid is what I would refer to as accidental models. From what I have seen, a large part of models developed by non-professionals are in fact accidental models.

An accidental model is what happens when you set out without a plan. When your purpose is simply to come up with something which makes money. Throw some indicator together, tweak settings, run optimizers, switch around indicators, values and instruments until, presto, you have got yourself a backtest that shows strong returns.

It's not all that difficult to build a backtest that shows great returns. The trick is to find predictive value going forward. If you just experimented with

settings until the results looked good, all you have done is fitted the algorithm to the known data. That has no predictive value and is highly unlikely to continue to yield attractive returns on real life data, going forward.

A proper trading model needs to start off with a theory about market behavior. It needs to have a clearly stated purpose in what market phenomenon it's trading. A *raison d'etre* .

I have to confess that when I was first told about this idea, I thought it was hogwash. Whether I read it or was told, I do remember that it was sometime in the mid 90's. The question was put to me on what I believe about the market. It sounded like total nonsense. After all, all I believed about the market was that I could get rich quickly if I just figured out the right combination of indicators and settings for a trading system. The idea that I would somehow have a theory about exploitable market behavior seemed more than a little far-fetched at the time.

No need to worry if your initial

reaction is the same. You will figure it out.

There are two common ways of looking at model purpose. One way may seem surprising for those who have not yet worked in the financial industry.

The first way is fairly straight forward. What you might expect. You start off with a theory of some sort. Perhaps something you have observed in the market, or something you read about. Now you want to test if it really works, and you formulate mathematical rules to test that hypothesis. This is how most successful trading models start out.

The second and perhaps surprising way is based on a perceived need or business opportunity. Someone working full time with developing trading algorithms may not have the luxury of dreaming up anything he or she wants. You may have a specific brief, based on what the firm needs or what it thinks the market may need.

That brief may for example be to construct a long only equities model,

where holding periods are long enough to qualify for long term capital gains tax, while having reasonably low correlation to existing equity strategies and have a downside protection mechanism. Or perhaps the brief is to study a type of strategy where competing asset management firms seem to be expanding and see if we can join in the competition for those allocations.

Often the return potential of a trading model may be of relatively low importance. The purpose may simply be to achieve a near zero or negative correlation to a currently used approach, while being able to scale to hundreds of millions, and preferably showing a modest positive expected return of a couple of percent per year. A model like that can greatly improve diversification for a large firm, and thereby enhance the overall long term performance of the firm's asset.

In particular at larger quant trading firms, model briefs are likely to start out with a business need. It's not a matter of finding a way to generate maximum return, as that rarely makes business

sense.

The concept of starting from scratch with no specific requirements and just coming up with a model that makes the most amount of money is something very rare. This is a business like most others. In the auto industry, it wouldn't make sense for everyone to attempt to make a faster car than Bugatti. There is greater demand for Hyundai style of cars.

Either way, you need to start out with a plan, before you start thinking about trading rules or data.

Rules and Variations

Generally speaking, you should aim for as few rules as possible and as few variations as possible.

Once you have arrived at a model purpose, you need to figure out how to formulate this purpose in terms of trading rules. These rules should be as simple and as few as you can muster. Robust trading models, those that work over the long run, tend to be the ones that keep things

simple.

Any complexity you add needs to pay off. You should see complexity as something inherently bad, something which needs to justify its existence. Any complexity you want to add to your model needs to have a clear and meaningful benefit.

Moreover, any complication or rule that you add needs to have a real life explanation. You can't just add a rule just because it seems to improve backtest performance. The rule needs to fit into the logic of the model purpose and play a clear rule in achieving that purpose.

Once you have arrived at a set of rules for testing your market theory, you probably want to try some variations. Note that there is a world apart between testing variations and optimization.

As an example, let's assume that you want to test a mean reversion type of strategy. You believe that when a stock has fallen four standard deviations below its 60 day linear regression line, it tends to bounce two standard deviations up

again.

Now you already have multiple parameters in play. Modeling and testing these rules is a fairly simple task. You could try a few variations of this, perhaps to expect the bounce by three or five standard deviations, using 30 or 90 day regression or a variation in the target bounce distance.

Making a few variations like this can be useful, both for testing parameter stability and to actually trade some variations of the rules to mitigate over-fitting risks.

What you don't want to do is to run an optimizer to figure out that the optimal entry is at 3.78 standard deviations, on a 73 day regression, using a target of 1.54 standard deviations. Such data is absolute rubbish.

Optimizers will tell you what the perfect parameters was for the past. They will also con you into a false sense of security, and make you believe that they have any sort of predictive value. Which they don't.

No, skip the optimization. But make a few variations of the rules, using reasonable, sensible numbers.

Handling Data

The process for how to use data for developing trading strategies, testing strategies and evaluating them is a controversial subject. It's also a subject which deserves books all by itself, and this book does not aim to go into any real depth on the subject.

A few things are important to understand in this context. Most important is to understand that the more you test strategies on a set of time-series data, the more biased your test will be. Whether conscious or not, you will be fitting your model to past data.

A simple example of this would be handling of 2008. If you are developing long equity models, you will quickly realize that what seemed to work great up until 2007 will suddenly show a massive drawdown in 2008. That was a pretty

eventful year, and if there are readers here who are too young to be aware of it, all I can say is lucky you.

So now you probably just slap a filter of some sort on there to avoid this horrible year. That filter may have reduced profitability in earlier years, but in the long run it paid off.

This would be a great example of Brownian motion. No, not that sort. As in Doc Emmet Brown. As in time travel. No, I'm not going to apologize for that gag, no matter how bad it may be.

Adding a specific rule to deal with 2008 makes your backtests look great, but it may constitute over-fitting. The simulated 'track record', if you can call it that, will indicate that you would have performed amazingly during this exceptionally difficult year. But would you really?

Had the model been developed before that year, you would likely not have accounted for the possibility of a near implosion of the global financial system.

While there are various methods of alleviating risks of these sort of mistakes, the easiest is to use part of the data series for fitting and part of it for testing. That is, you only use a part of your time-series data for developing your rules, and when you are done you test it on the unused part.

This is a subject which I recommend that you dig into deeper, but also a subject which would take up too much of this book if I go into too much details. Besides, Robert Carver (Carver, Systematic Trading, 2015) has already written a great book which covers this subject better than I could anyhow.

Asset Class

There are different perspectives you can take when classifying asset classes. It would be perfectly valid for instance to say that the main asset classes are stocks, bonds, currencies and commodities. For most market participants, that way of looking at asset classes makes the most

sense.

But for systematic, quantitative traders, another definition may be more practical. When looking at the various markets we have available to us, we can group them in different ways. One way to group asset classes would be to look at the type of instruments used to trade them. The type of instrument is, for a systematic trader, often more important than the properties of the underlying market.

This becomes particularly clear with futures, as we will soon see, where you can trade just about anything in a uniform manner. Futures behave quite differently than stocks, from a mechanical point of view, and that's important when building trading models.

The currency space is an interesting demonstration of this concept. You can trade spot currencies or you can trade currency futures. It's really the same underlying asset, but the mechanics of the two types of instruments is very different, and would need to be modeled in different ways.

For that reason, asset classes are in the context of this book based on the mechanical properties of the instruments.

We are mostly going to deal with equities and futures in this book. There are two reasons for that, which happily coincide. First, the backtesting software which will be introduced in this book supports only these two asset classes. Second, these happens to be the asset classes which I personally prefer and have most experience with.

Investment Universe

The investment universe is the set of markets you plan to trade. It's a very important factor to consider for your trading strategy. The assumption through this book is that you aim to trade a set of markets, and not just a single one. It's usually a bad idea to trade a single market and most professional grade strategies are designed as portfolio strategies.

If you would start off by picking a single market to trade, you have already

made the most important decision. When someone sets out to make a great model for capturing bull runs in the Dow Jones Index, he has already limited himself. Perhaps the strategy he designed is just fine, but this particular market may perform poorly for the next few years. No, diversification is the way to go. Apply your trading strategy on multiple markets, and your probabilities of success are much improved.

How you select your investment universe is of very high importance. What most people do, conscious or not, is to select markets that did very well in the recent past.

Investment universe selection works differently for different asset classes. Every asset class has unique issues and solutions, and we will look at specifics later on, in each asset class section of this book. One thing to keep in mind though, is that the greatest potential for catastrophic error in this regard lies in the equity sector.

Allocation and Risk Level

Allocation is about how much risk you want to allocate to something. To a position, to a trading model, to a variation of a trading model, to a portfolio etc. It's a much wider topic than simple position sizing.

Ultimately, the question you want to answer is how much of an asset you should be holding. The way you get to the answer can be quite complex, and there may be many moving parts in play.

When you consider what approach to allocation to take, you need to think if in terms of risk. By risk, I mean the way the term is used in finance. This is a topic which is all too often misunderstood by retail traders.

Chapter 4 will deal more with financial risk, and this is an important topic. If you want to move from hobby trading to the world of professionals, the most important point to understand is risk

and how it relates to allocation.

The models in this book will aim for risk levels which are generally considered to be in the acceptable range for institutional asset management. They will aim for attractive enough return to be worth bothering with, while keeping the risk profile on a level which could be used in professional setting.

If on the other hand, you are looking for something spicier, I would recommend that you take a look at (Carver, Leveraged Trading, 2019). Yes, that's the second recommendation for the same author so either I really like his books or I'm being coerced to writing this book while being held in Rob's basement for the past year and plugging his books is the only way out.

Entry and Exit Rules

This is the first thing that most people think of when designing a trading model. It's the most obvious part, but it's not necessarily the most important part.

Naturally any trading model needs rules for when to initiate a position and when to close it. For some types of strategies, the exact timing of these events can be of critical importance. But there are also strategies, often of longer time horizon, where the exact entry and exit points are of subordinate importance.

It wouldn't be fair to say that entry and exit rules are not important. Just keep in mind that they are not the only parts of a strategy that matters. Many portfolio based models rely more on what mix of positions you have at any given time than exactly when you opened them.

Rebalancing

The rebalancing part is an often neglected part of trading models. While not necessary for many shorter term trading models, it can have a significant impact on models with a longer holding period.

Rebalancing is about maintaining a desired allocation. If you would get

invited to look at the trade blotter for a systematic trading shop, you will likely see that there are a lot more trades done than you might expect. Even if the strategy is long term trend following, you may see that there position sizes are adjusted often, perhaps even every day. Small changes, up and down, back and forth, for no apparent reason.

You may see a long position opened in January one year and closed out in September. But in between those points, there may be a large amount of smaller trades, changing the position size up and down. You might wonder what caused the position to be increased or decreased. But that's not what happened.

These trades were rebalancing trades, aiming at maintaining the desired risk level. They were not changing the positon, merely maintaining it. Remember that most professional trading models aim to hold a certain amount of portfolio risk on a position. The risk calculation involves things like volatility of the instrument and the size of the portfolio. These things are not static.

As the volatility changes in a market, or your portfolio as a whole changes due to other positions, your position risk would change, and you would need to make adjustments just to maintain the same risk. That's what rebalancing is about.

Not every model requires rebalancing, and even if you decide not to employ it, you should still understand the concept and the implications of not rebalancing.

4 Financial Risk

Financial risk is about potential value variation per unit of time.

That sentence is of sufficient importance to deserve its own paragraph. This is a very basic concept and one that's absolutely clear to anyone working in the financial industry. If you have a university degree in finance or if you have spent time on the professional side of the industry, this should already be very obvious to you.

However, unfortunate as it is, the concept of risk is often very much misunderstood and misused in the hobby trading segment. A large quantity of books and websites targeting hobby traders has muddled the concept of risk and continue to misuse the term and promote methodologies based on gambling, numerology and pseudoscience.

In this chapter, I will attempt to explain what risk means in a financial context, how it can be measured and used for systematic trading models, as well as explain the common ways that the term is misused and the dangers it poses. But first, take a look at that key sentence again.

Financial risk is about potential value variation per unit of time.

Quantifying Risk

The most common way to quantify risk is about measuring past volatility. The term volatility has to do with how much an asset tends to fluctuate, move up or

down, in a given time period. You may for instance conclude that while Microsoft tends to fluctuate by about half a percent per day on average for the past few months, Tesla shows daily changes of double that.

That would mean that Tesla is more volatile than Microsoft, but whether or not it's more risky depends on how much you invest in it. If the volatility is exactly half in Microsoft, you would theoretically achieve the same risk level if you invest twice as much as in Tesla. You would, again theoretically, see the same value variation in each stock per day going forward.

That's in a nutshell how to think about financial risk.

Before I offend actual financial risk professionals too much, in particular as I'm married to one, I should point out that financial risk can be an extremely complex topic. If you are a risk manager or risk controller of a larger portfolio, of a number of portfolios or something as complex as a whole bank, risk can be very

mathematically intensive. But that's not what this book covers. This book covers risk as it pertains to trading a single, or at least a small number of portfolios.

The most important part to understand from that perspective is that risk, just like return, always has a time component. If you are offered a ten percent return, you can't possibly say if that's good or bad without knowing about the time frame. Ten percent in a month, ten percent in a year, or ten percent in ten years are all very different situations.

The same naturally goes for the risk side. Risking a loss of two percent in a day is not the same as risking a loss of two percent in a year. This is why we use historical volatility of assets, measuring how much they move up or down per day, to calculate risk exposure.

If you understand this concept alone, that risk has to do with potential value variation per unit of time, you have come a long way from the hobby trading way of thinking.

As always, there are of course

potentially complicating factors to this otherwise simple concept. While I will refrain from going into these in great detail, as they could easily warrant a book on by themselves, I will briefly explain what it's about.

The first issue has to do with correlation. This should be considered when you have multiple positions open, and has to do with how they relate to each other. If you hold two positions in two similar stocks, with a high correlation to each other, you are adding to the same or a similar risk factor.

If on the other hand, you hold a stock and a commodity, they could potentially be completely unrelated, and holding them both could be lower overall risk than only holding one of them. No need to worry if this seems overly complicated to you. For the purposes of this book I will keep things simple and not dive into the fun sounding topic of covariance matrix analysis.

The other issue with using recent volatility as a forecast for future volatility

is that this relationship does not always work out. Meaning, that sometimes future volatility can be quite different from the past. Or in plain English, sometimes things change.

But on the whole, using past volatility to estimate future volatility is better than not doing so.

This concept of volatility can also be applied to the past, which things are bit less in in a state of flux than in the future. If you evaluating the past performance of trading portfolios as an example. It's not enough to see that Portfolio A showed a higher performance than Portfolio B. You need to put it into context of what risk they took to get there. How high the volatility of returns were.

This is why the Sharpe Ratio remains one of the most commonly used comparison measurements. This ratio takes the overall returns, deducts risk free interest for the same period, and divides this by the standard deviation.

Mark to Market

The term mark to market is a principle of valuing something at the most recent market price, taking all known factors into account. This may seem like an obvious concept, but is often overlooked by those lacking financial education or market experience.

Your holdings as well as your overall portfolio should always be valued at their current mark to market valuation. That's not anywhere near as complicated as it may seem. It's best explained by an example of someone breaking this rule.

People seem to like gambling analogies. A guy walks into a casino and puts down a hundred dollars in chips on the black jack table. After a good run, he now has two hundred dollars in chips in front of him. In his mind, he is now playing with the bank's money. He could lose a hundred dollars before risking anything.

That's now how it works though. After he doubled his money, his mark to

market portfolio is $200. If he now loses ten dollars before leaving the table, he walks away with $190. That's a gain from when he sat down, but a loss from his peak mark to market valuation.

Now look at the trading equivalent. Someone buys 500 Initech shares at $20, with a plan to lose at most $1,000 on this trade. The share takes off right away, and moves up to $30. Now your initial position value of $10,000 is up to $15,000. Then the price starts moving down for a while. Finally it hits $18, and he stops out with a portfolio value of $9,000, for a loss of $1,000.

But that's again not how it works. The position was at one point worth $15,000, so he lost $6,000 from there.

From a valuation perspective, it does not matter if you have closed your position or not. The value of your position, and in extension of your portfolio, is not affected by whether or not you still have an open exposure. The current value is what counts, and the current value is based on the last known

market price of the shares, if the position is open, or the cash proceeds if it was closed.

To understand mark to market, you need to think in terms of state. Consider the current state of your portfolio. What it's worth right now, at this moment.

Common Risk Fallacies

It was no accident that I used a gambling analogy in the previous section. Gambling analogies are very common in hobby trading literature and unfortunately so are gambling approaches to risk. Such approaches can not only be very dangerous, they also tend to be quite nonsensical and disregard basic economics.

These methods are usually collected under the umbrella term Money Management. This is a term that you won't likely hear in the financial industry, but which seems quite common in the

hobby segment.

A popular idea in this space is to apply what is called position size pyramiding. This premise is that you increase a position upon success, as you are now playing with the bank's money, and thereby your risk is lower.

But as you saw earlier, this defies the most basic idea of what risk is and what it means. I will also demonstrate here why it just does not make any sense.

A trader buys 1,000 shares of Pierce & Pierce at $10. A few days later, the price has moved up nicely and is now trading at 20. At that point, our fearless trader decides to double his position, and buy another 1,000 shares. The price moves up and down a bit, but some days later the trader finds the stock at $25, and he buys another 1,000 lots.

This is the general idea of pyramiding. To increase position on success. And it also makes no sense.

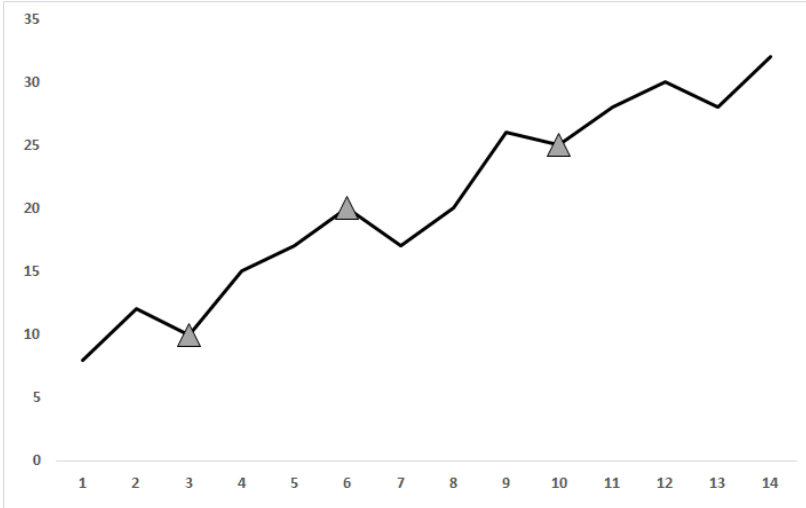


Figure 4-1 Pyramiding

In this simple example, we don't really know why the trader decided to buy exactly 1,000 shares, but that was the starting position. A few days later, he doubled his risk, simply because the first entry point turned out to be favorable.

The problem with that is that your past trades lack a magical ability to impact the future. The fact that you bought at 10 does not in any way affect the probabilities of prices moving up or down from here. Yet the trader decided to double his position.

Ask yourself what you would have done if you missed that first entry at 10

for some reason. When you came back to see the screens, the price was already at 20. What now? Would you buy 1,000 shares? 2,000 shares? Or pass on the trade?

That's an important question. This goes back to the importance of thinking about states, and your answer to that question will show if you follow financial logic yet or not.

If you had taken that first trade at 10 and now doubled, as per the original plan, you would now hold 2,000 shares. From a logical point of view, if you missed the first trade, you should now buy the full 2,000 shares. That's the only way that you end up with the same portfolio state as originally intended.

If your answer is that you should only buy 1,000 or skip the trade, then you seem to believe that your own past trades can magically impact the future. And there really lay the problem with pyramiding, and with most other so called "money management strategies". They are based on common gambling fallacies that

have no grounding in logic, mathematics or finance.

A similar idea is the concept of *Risk per Trade* . The first time I was asked how much I risk per trade, I was quite puzzled. I couldn't figure out what that question meant. Risk per trade and day?

This is again a dangerous fallacy. Risk per trade is the notion of risk being how much money you would lose if an imaginary stop loss point is hit at some time in the unspecified future. That way of thinking about risk is just plain wrong. There is really no other way to look at it. This way of defining risk is simply a misunderstanding of what the word means in a financial context.

Take two portfolios as an example, each with a million dollars to start out with. We are going to buy IBM shares, and only IBM shares and those are right now trading at $180. For Portfolio 1 we are buying 4,000 shares, for a total notional exposure of $720.000. With Portfolio 2, we only buy 2,000, and thereby have half of the exposure, $360,000.

We set our stop loss point for Portfolio 1 at 170, and the stop loss point for Portfolio 2 at 155. Which portfolio is more risky?

Table 4.1 "Risk per Trade"

If your answer was Portfolio 2, you need to rethink how you view risk. This is a very important issue, and it's really a problem in how hobby trading literature is obscuring the subject.

According to this odd risk-per-trade way of reasoning, Portfolio 1 can lose up to $40,000 if the stop loss point of 170 is hit, while Portfolio 2 can lose up to $50,000 if the stop at 155 is hit. But this is not what risk is. Obviously, Portfolio 1 is twice as risky as portfolio 2.

If IBM falls by one dollar tomorrow, Portfolio 1 will lose $4,000 while

Portfolio 2 will lose $2,000. Risk always has a time component to it. Just like return does.

Risk as Currency to Buy Performance

Performance always has to be put into the context of risk taken. The game is never about who has the highest return in a year. It's about who has the highest return per unit of risk. A gambler putting his entire fortune on number 17, spinning the roulette wheel and winning, made a lot of money. The risk however was rather extreme.

One of the most important steps to becoming a professional trader is to understand what is possible and what is not. Many people entering the field have a belief in seemingly magical return numbers at little to no risk. Unfortunately, there is a thriving industry of con men who are happy to help sell such dreams. They all have colorful stories of how they

made millions in short periods of time and how they figured out the secrets of the market. And then they decided to go into the business of mentoring or system selling.

If you are aiming for triple digit yearly returns, you are in the wrong field. It just does not work like that. Nobody has done that. You will not succeed.

In a single year, anything can happen. Anyone can have an extremely good year now and then. But to expect to compound at triple digit numbers over time is the equivalent of attempting a hundred meter dash in two seconds.

Imagine for a moment that the con men are correct. That if you pay them for their trading systems and mentoring, you can achieve 100% consistent return per year. What would that mean? The math is quite simple. Sell your car and put $10,000 to work. In a year, you will have 20k. In two years you have 40k. In ten years, you will have ten million dollars. Twenty years after you started, you will be the proud owner of ten billion dollars

and you would see your first trillion in year 26. It would be nice if such fantasies worked, but as it turns out, magic is not real and, spoiler, neither is Santa.

Anyone aiming at achieving triple digit yearly returns will, with mathematical certainty, lose all of their money if they remain at the table. In such a game, the longer you play, the more your probability of ruin approaches 1.

To put things into context, some of the very best hedge funds in the world show real world compound returns of around 20% per year. And those are the best of the best. This is the league that Buffett and Soros are playing in.

So what can we expect?

The first thing to understand is that the higher risk you are willing to take, the higher will your possible returns be. Just like the guy at the roulette table, you can make enormous gains if you don't mind having a very high probability of losing all of it.

Knowledge, hard work and skill can

improve your results but it can't accomplish the impossible.

The bad news is that your returns are likely to be less than 15% p.a. in reality over longer time periods. The good news is that if you can achieve that, you can make a lot of money in this business. But of course, performance need to be paid for, with volatility.

Nobody likes volatility. It would be great if we could get a small gain every single day, and move up in a straight line. But unfortunately, volatility is required to create returns. What we try to achieve is to use as little volatility as we can to pay for the performance. And that's what a Sharpe Ratio measures.

Sharpe Ratio is probably the most widely used and most well-known performance metrics. It can be a very useful analytic and it gives you a general idea of the risk adjusted performance of a strategy. Naturally you need to go deeper and analyze the details for a proper strategy evaluation, but the Sharpe will give you a good overview to start off with.

The calculation of the Sharpe Ratio is quite simple. You take the annualized return, deduct risk free interest and divide by annualized standard deviation of returns.

$$Sharpe = \frac{AnnualizedReturn - RiskFree}{AnnualizedStandardDeviation}$$

The part of this formula which tends to raise the most questions is the risk free rate. The proper way to do it, at least in this author's view, would be to use a shorter money market or treasury yields. That is, a time series of yields, deducting the daily yields from the daily strategy returns, and not just a fixed value.

But this is a practical book, and I will give you a practical advice. For most readers of this book, the risk free rate aspect may seem like an unnecessary complication. If your purpose is simply to compare strategies against each other, you might want to take the shortcut of using zero as the risk free rate.

Given the formula, we clearly want to see a high Sharpe Ratio rather than a low one. We want a high return at a low

volatility. But you need to be realistic here. What you will find is that Sharpe Ratios over 1.0 are rare, and that's not necessarily a problem.

Some strategies can be highly successful and very profitable, while still showing a Sharpe of 0.7 or 0.8. A realized Sharpe of above 1.0 is possible, but exceptional. It's in this type of range that is makes sense to aim.

Strategies with Sharpe of 3 or even 5 do exist, but they tend to be of the so called negative skew variety. That expression, referring to the shape of the return distribution, means that you win small most of the time, until you suddenly get hit with a big loss. For such strategies, you may see long periods of constant winning, only to suffer a sudden and sometimes catastrophic loss.

A Sharpe Ratio does not tell the whole story, and by itself it shouldn't be used to evaluate, select or discard a strategy. It does make sense to use in combination with other analytics and a detailed analysis.

What makes the Sharpe Ratio so useful is that it directly reflects the core concept that this chapter is trying to instill. The concept that returns always have to be put in context of volatility. By understanding the logic of the Sharpe Ratio, you will gain an understanding of the concept of financial risk.

5 Introduction to Python

So here you are. You bought a trading book written by that guy who tends to keep things simple. You got sucked in by his last two books. They both followed a simple formula, explaining a single strategy each. One simple concept, slowly explained in detail over a few hundred pages in a way that anyone can understand. And now you are starting to realize that you got tricked into buying a programming book and it's far too late to ask for your money back.

Well, now that you already paid for

the book, you might as well stick around and learn something. Yes, I'm going to teach you some programming in this book. No, it's not going to hurt. Much. It won't even be all that difficult. Strap in and make yourself comfortable. You are in for a long ride.

Some Assembly Required

I have been using computers since before many of you readers were born. And that's a painful thing to say. Back in the 80's, computers were mostly dismissed as useless toys. In all fairness, they were pretty fun toys. Even in the early 90's, nobody would take computers seriously. These were the days when nerdy Scandinavian kids like me would program little 'demos', with jumping and blinking text that proclaims how smart we are in the coolest American street slang as we could muster, saving it on 5.25" floppy disks along with perfectly legal copies of computer games, put them in paper

envelopes and mail them around the world to like-minded individuals. It was a strange time.

The seismic shift happened in late 1995. It was actually quite painful to watch for us computer guys. At the time, I was a member of both the university computer club, and the main party frat. That was not how things normally worked, and I was careful in keeping the two as separate as possible. But when the president of the frat came up to me in late 1995 and asked if it's worth upgrading his 9600 bps modem to a new 14.4k, that's when I knew that our world was over. The barbarians were at the gate and there was nothing we could do to hold them back.

My premonition turned out all too true, and much like Cassandra I was helpless to change it. Suddenly everybody and his literal grandmother thought computers were the coolest thing ever. In 1994 email was something for a small group of computer enthusiasts. In 1996 your grandmother had a home page. It was a traumatic experience.

But it was not all bad. Suddenly even the most elementary computer skills became negotiable assets. Computers now became an integral part of almost any profession. You just couldn't get by without being able to use email, Word or Excel. It's almost hard to imagine now, but in 1994 few people in the world had any idea of the concept of files and folders (though we called them directories back then).

I'm sure you are all wondering if there is a point to this walk down memory lane or if I simply got old enough to rant on. Believe it or not, there actually is a point here.

Most people today view programming as some sort of specialized task for propeller heads. Programming is for programmers. The kind of people who do nothing but program all day.

Very much like how people once saw typing as a difficult and menial task, reserved for secretaries. Or how the idea of learning about file systems seemed outlandish to people in 1994. Yet today,

you would have a severe disadvantage in most skilled professions if you are unable to type at reasonable speed and even more so if you don't understand basic computer usage.

You don't have to become a programmer. There will always be people who are much better than you at programming. But that does not mean that you should stay ignorant on the subject.

Programming is not as difficult as most people would think. Once you start learning, it can be incredibly gratifying to see the results on the screen. And you don't need to go into any real depth.

Once you learn some elementary programming, you can get things done by yourself. You no longer have to rely on specialists to do these things for you. Just like how you don't need to dictate a letter for your secretary to type anymore.

So if we are over the technophobia now, let's move on.

Python Emerges as

the Logical Choice

From a finance and trading point of view, Python as a programming language is objectively special. It's not just yet another language with some different syntax and minor differences. It's a potential game changer and something you should really pay attention to.

Python is very easy to learn. Whether you are new to programming or a seasoned C++ coder, you can get into Python very quickly. The syntax is deliberately very easy to read. If you know nothing about Python and are shown a bit of code, you will right away see what it does. That's not true for most programming languages.

Python is to a large extent purpose built for finance. There are tools available which are designed by hedge fund quants and made available to everyone for free. Tasks which would take a lot of programming in a C style language can often be done in a single line of code. Having programmed in many different

languages for the past 30 years, I have never seen a language where you can get things done as quickly and easily as in Python.

Python is an interpreted language. That means that you don't compile your code into binary files. If those sentences don't mean anything to you, there is no need to worry. It's not important.

The **exe** and **dll** files that you see on your computer, should you be using Windows, are compiled. If you open them in a text editor, you get seemingly random garbage displayed. Compiled code is faster than interpreted code but not as easy to build and maintain. The interpreted code on the other hand is just a text file. It's translated on the fly, as it's being executed.

In the past few years, Python has emerged as the quant's language of choice. This has resulted in a substantial community and a large amount of open source tools. For people working in finance, the quant community are surprisingly open to sharing stuff.

There are some clear issues with Python of course. Given that most people using Python at the moment are hard core quants, documentation tends to assume that you already know everything and there is a clear aversion to anything user friendly. For someone entering this space, the first barrier to overcome is what at times look like a certain techno arrogance.

Most people working with Python do almost everything at a text prompt, occasionally spitting out a simplistic graph but mostly just text. There is really no technical reason for the lack of graphical environments, but it's rather a cultural thing.

With this book, I hope to make Python more accessible for traders. It really is a great tool. Don't let them scare you away.

Programming Teaching Approach

As opposed to most actual

programming books, I won't start off by going over all data types, control structures and such. You will get the hang of that later on anyhow, and it's not terribly important from the very start. In fact, I think that the usual way that programming books are structured tend to scare off, or bore off, a lot of readers. Besides, there are already plenty of such books out there which are written by people far more competent than I in explaining the in-depth technical aspects.

I will instead take a little different tack. I'm going to just drop you in it and let you swim. This is a practical, hands on book and my aim here is to get you up and running as quickly as possible. Learning about the specific differences between a tuple and a set is not important from the get go, and as we move along in this book you will pick up sufficient knowledge to get the important tasks done.

That also means that I won't explain every different possible way of doing something. I will pick one which I think will be helpful for you at the moment.

Most of the time, there are multiple ways, methods, libraries or tools that can be used. I won't explain them all in this book. I will show you one path of getting things done, and once you feel confident enough to explore, you will see that the same task could be done in many different ways.

I will start by showing you how to install and set up Python on your computer. Then we will play a bit with Python, trying some code out and getting a feel for how things work.

After this, we are going to install a Python based backtesting engine. That's what we will use to run the simulations in this book. At first, we are going to do some basic, simple simulations. As the book progresses, we will get to increasingly complex and realistic modeling, and my aim is that you will get more and more comfortable with working with Python backtesting over the course of this book.

Working with a language environment like this, there is always a

risk that something will change. That new versions are being released after this book, and that some parts simply don't work as they should anymore. I will do my best to reduce such risks, but it can't be avoided all together. Should such issues come up, take a look at my website for updates and explanations.

Installing Python on your Computer

Python can be run on many types of computers. It can even be run via some websites, where all is managed on a server by someone else. That can be a good idea for some things, but in this book we are going to run everything locally on your own machine. That way, you will have full control of your code, your data and your environment.

For the purpose of this book, it does not matter if you have a Windows based computer, Apple or Linux. I'm using Windows, and that means that screenshots

will be from that environment, and there may be minor differences here and there in case you are on a different operating system. But it shouldn't matter much.

I don't see a reason to proclaim any of these operating systems somehow better or more suited than another. I use Windows as some financial software on my machines are only available for Windows, and because I prefer to work with Windows. In the end, it's just a tool. It would be silly to have emotional reasons for picking a tool. Pick one that gets the job done.

An enormously helpful software package for Python is **Anaconda** . That's what we will use for much of this book. It's a free software package and it's available for **Windows** , **MacOS** and **Linux** .

Anaconda is the de facto industry standard software package for developing and testing with Python. It's actually a collection of programs which are all installed when you download the main **Anaconda** package.

With **Anaconda** , you will get some nice graphical tools to work with, so that we don't have to do everything at a command prompt. It just makes Python life easier.

Head on over to the **Anaconda** website, (**https://www.anaconda.com/download/**) and download the latest package for your operating system. You will see that they give you a choice of downloading Python version 3 or 2. Go with the former.

Python 2 is now quite old, and there is very little reason to use it. Unless you have a really good reason to pick that one, go with version 3.

Let's Run Some Code

There are a multitude of different applications and environments where you could write and run your Python code. Two of the most common are already installed on your computer at this point, as they are prepackaged with the **Anaconda** installation.

I want to briefly mention the useful but annoyingly spelled application **Spyder** , before moving into more detail about the equally annoyingly spelled **Jupyter Notebook** environment which we will be using for almost the entire remainder of this book.

Spyder looks and behaves very similar to what most would expect from a programming environment. If you have prior experience with writing code, you would likely feel quite at home here.

The **Spyder** environment is great for working with multiple source files, creating or editing Python files and building libraries of code. As you get more comfortable with Python, I very much encourage you to look closer at this environment. I find **Spyder** to be a useful tool and it complements the Jupyter environment well.

The only reason that this book is not using **Spyder** is that many readers are likely brand new to both Python and programming, and I would like to avoid the additional confusion of having two

separate applications with different behavior to worry about.

For the purposes of modeling trading strategies, I find the **Jupyter Notebook** superior. In a generalization that I'm sure some experienced coders will take issue with, I would say that **Spyder** is great for serious programming while **Jupyter** is great for tinkering and testing stuff.

Since building trading models is very much a process of tinkering and testing stuff, we will be using Jupyter exclusively in this book.

When you installed **Anaconda** , a few different programs were actually installed at the same time, all part of the **Anaconda** package. Open up the program **Anaconda Navigator** . This is the hub, the main control center for all your new Python tools. We will be using this program more in this book and I'm sure you'll find it quite useful.

When you open up **Anaconda Navigator** , you should see something similar to Figure 5-1, where a few

applications in the **Anaconda** package are shown. As you see, both **Jupyter** and **Spyder** are listed here, among others.

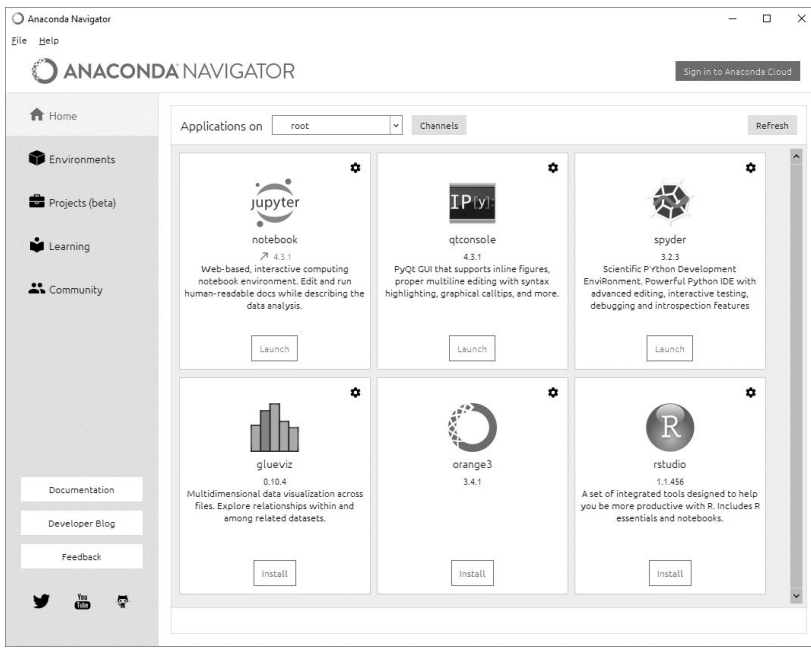


Figure 5-1 Anaconda Navigator

Click the launch button for **Jupyter** and see what happens. You might be surprised to see that a web browser is launched, and that you now have a web page in front of you, listing the files and folders of your user folder. What you will see is something very similar to Figure

5-2.

Believe it or not, this seemingly odd web page is actually a powerful tool which we will use to build and test serious trading models in this book.

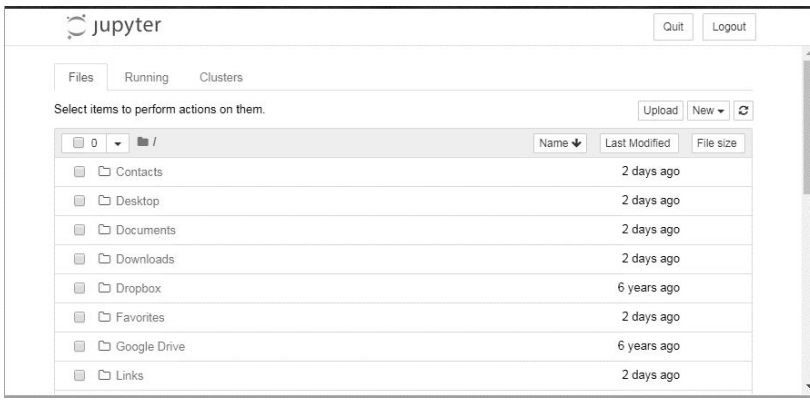


Figure 5-2 Jupyter Notebook

We are going to create a new file where we will write our first code. To keep things neat and organized, you may want to create a new folder for it first. You can create both a folder and a new Python file through that dropdown on the right side, which you can see in Figure 5-2.

After you make a new file, you will be met by a new web page, which likely looks like Figure 5-3. That text box there, just after the text `In [ ]:` is called a cell. In

a notebook like this, you write code in cells before executing, or running the code. As we have not yet named this notebook, the text Untitled can be seen at the top. You can click on that and rename your notebook if you like, and its file name will update automatically.

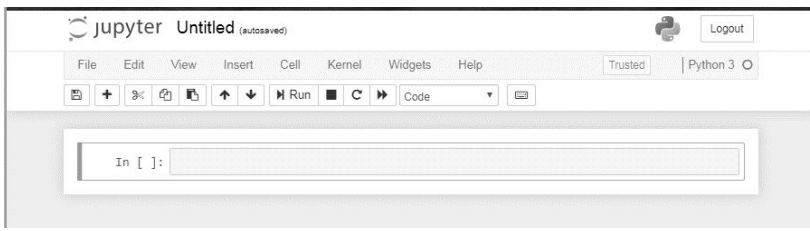


Figure 5-3 Empty Jupyter Notebook

It's time to write and run your first piece of Python code. This is a simple piece of code, designed to teach a few important concepts about this particular language.

Click in the cell in your new notebook, and enter the following code.

```
people = ['Tom','Dick',"Harry"]
for person in people:
   print("There's a person called " + person)
```

Now run the code. You can run it directly inside the notebook, either by

clicking the run button in the toolbar, or simply hitting **ctrl-enter** or the equivalent if you not on a Windows platform. The code will run right away, and the result output will be shown just below your cell, just like in Figure 5-4.

The output from this should, somewhat predictably, look like this.

```
There's a person called Tom
There's a person called Dick
There's a person called Harry
```

As you see, it's quite easy to look at a piece of Python code and immediately see what it does. That's one of the really great parts of this language, how easy it is to read. But now look closer at this little code snippet and see what it really does.

The first row, `people = ['Tom','Dick',"Harry" ]` , defines a list. Lists are pretty much what other languages would call arrays; simply a list of stuff. Note that we don't need to tell Python what kind of variable type to create. In case you are used to other languages, you may notice that there is no need to declare variable and type. Another great help, and one less

headache less to worry about.

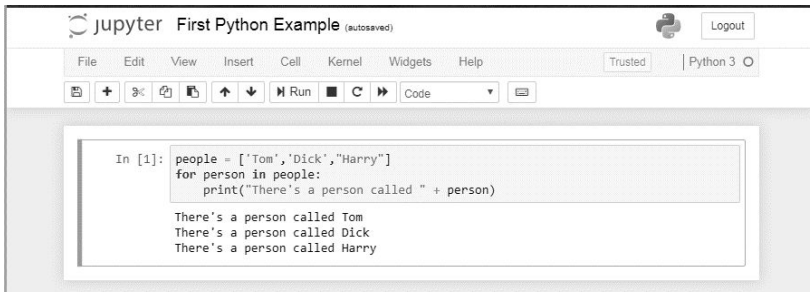


Figure 5-4 Your first Python Code

In the second row, `for person in people :` , we initiate a loop. This is also much easier than in most other languages. We just tell the code to loop through all the items in the list `peopl e` . First note the colon at the end of that row. This means that we are about to supply a block of code, in this case to be looped through. The colon introduces a block of code.

Think of it as a way of saying, “Now do the stuff that I’m about to tell you about below”. What follows below is a block of code. The instructions of what to do in this loop. You will see a very similar syntax later when making conditions, such as telling that code that if something is equal to something else, then run a block of code.

The next important point is how a block of code is defined in Python. This is a very important concept in Python. Note the indentation.

Many languages group blocks of code by using curly braces. Python does not do that. Here, a block of code is defined by the level of indentation. That is, the amount of blank space there is to the left of the text. Text on the same level, with the same distance to the left edge, are a group.

So the tab in on the row that starts with `prin t` , is absolutely necessary. That shows that this is a different block of code, subordinated to the previous one. You will see this concept all the time, throughout this book.

In this simple example, we had only one operation that we wanted to loop. The printing of a few rows of text. If we wanted to do more things, we could just add further instructions, more rows below the print statement, on the same level of indentation.

You may have noticed that I seem

to have a weird mix of single quotes and double quotes in that first row of code, `people = ['Tom','Dick',"Harry" ]` . Unless of course my copy editor decided to clean that up. I purposely used a really ugly syntax, where some strings are enclosed in single quotes, while one is in double quotes. I did that to show that it does not matter. You can use either single or double quotes.

But the reason I mixed the type of quotes in the print statement, `print("There's a person called " + person )` is to demonstrate something else. As you see there, I'm trying to print out a single quote that is part of the sentence. In that case, the easiest solution is to simply enclose the entire sentence in double quotes.

The `prin t` statement concatenates a text string with the name of each person and outputs to console. So from this little silly code sample, you have now learned about lists, loops, indentions, quotes and console printing.

If this in any way seemed complicated to you, then take a moment

and try this code out. Change it a bit. Make different types of lists, print different things. The best way to get a hang of it, is to just try it out for yourself.

Working with Jupyter Notebook

You saw in the previous example that in a Jupyter Notebook, we write the code in a cell, and that when it is executed, the result is shown right below that cell. A notebook can have many cells, each with its own code. Sometimes it can be very useful to break the code up into a few different cells, to make it easier to keep track of, and avoid having to re-run all of the code if you change a small part. It makes tinkering easier.

What's important to understand is that the cells in a notebook share a common name space. What that means is just that each cell is aware of the results of the others. Once you have executed a code in one cell, you can refer to variables

from that in other cells.

To test this concept out, make a new cell in the same notebook, below the one where we played with the list before. You can just press the plus sign in the toolbar and you will get a brand new cell.

In this new cell, write this code.

```
print("The first person is " + people[0])
```

With this row, I want to show two things. The first thing is that this second cell is aware of the variable peopl e which we created in the first cell. And as a second point to learn here is that lists are zero based. That is, we are grabbing the first element of this list with the syntax people[0] . As the list is zero based, the first element is 0, the second element is 1 and so on.

Figure 5-5 demonstrates how this should look, and how the code output would appear. The idea of working with cells like this, writing and executing code segments in the cells, is something that you will have great use for when tinkering with Python.

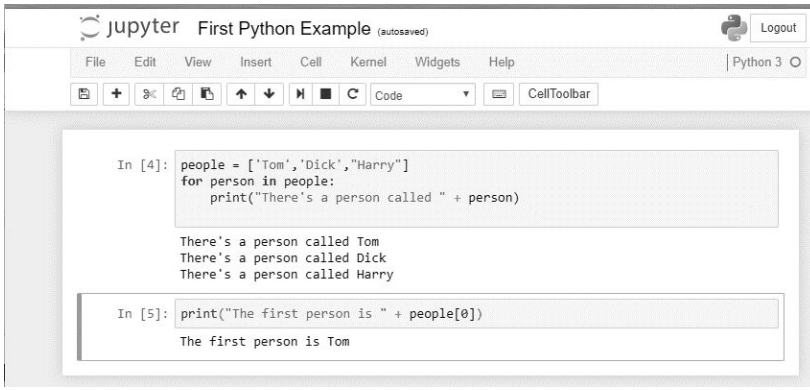


Figure 5-5 Working with Cells

Dictionary Lookup

Two very closely related concepts that you will come across all the time are lists and dictionaries. In the previous section, you saw an example of how lists work. Lists are just what they sound like. Lists of just about anything. A dictionary on the other hand, is a lookup table which matches two items with each other. Just like, well, an actual dictionary.

As you just saw above, a list is defined using square brackets, like this `[one, two three ]` . A dictionary on the other

hand uses curly braces. This makes it easy to tell in the code if you are dealing with a list or a dictionary.

Giving you a feel for what a dictionary is and how it works, let's try a brief example. We can create a new dictionary using curly braces, like the list of tickers and matching company names below.

```
stocks = {
      "CAKE":"Cheesecake Factory",
      "PZZA":"Papa John's Pizza",
      "FUN":"Cedar Fair",
      "CAR": "Avis Budget Group",
      }
```

The code above, defining the dictionary, is just one line of code. As you see, it's possible to break the line up to make it easier to read, as I have done here. A dictionary defines pairs of items, separated by a colon, in this case a ticker followed by a name after the colon.

Another thing you might notice is that I left the comma after the last item set. Clearly that's not needed, as nothing comes after it. I left it there to show that it doesn't matter. As opposed to most other

programming languages, Python won't complain about this. Just another way that Python is easier and simpler.

We can now look up values in this dictionary, or we can loop through and get all the items one by one. To look up the company name behind ticker PZZA we just write stocks[“PZZA”] .

```
print(stocks["PZZA"])
```

Or if we want to iterate the items and print them all, we can use this kind of logic below.

```
for ticker, name in stocks.items():
   print("{} has ticker {}".format(name, ticker))
```

This is a clever thing in Python that you will see more of. We can unpack the item pairs, and get both the ticker and the name each loop through.

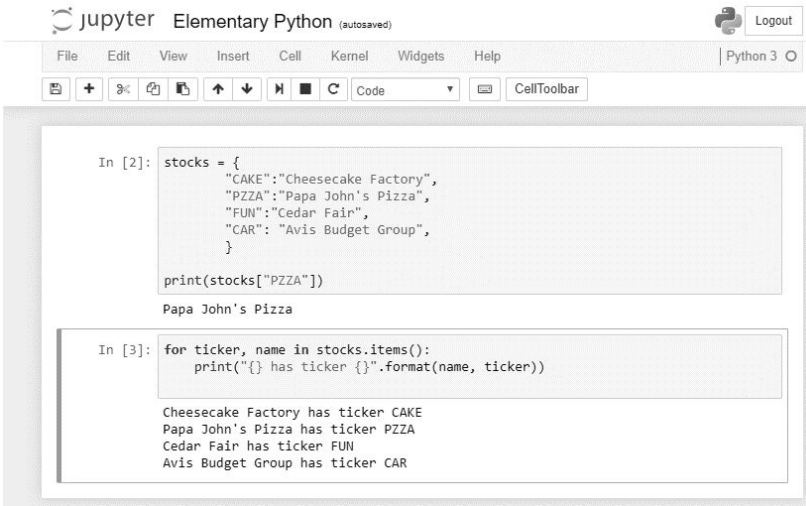


Figure 5-6 Working with Dictionaries

I also used a useful Python way of constructing a text string here. Previously we simply used plus signs to construct strings, but this way is so much easier. Using plus signs, we could have concatenated the string like this:

```
name + " has ticker " + ticker
```

That works just fine, but it's much more convenient to just write the entire string first, and then have the variable values inserted into it at the right spots. This way also has the advantage of allowing us to make formatting choices, for number values for instance, which we will do later on.

```
"{} has ticker {}".format(name, ticker)
```

That will result in the same string, but will be easier to manage. The curly braces in the text string show where we want variables to be inserted, and the function format( ) will handle it for us. We can insert as many or as few variables as we like into a text string this way.

Possible Issue Starting Jupyter

Based on the first edition of this book, a reader reported issues trying to start Jupyter on his local machine. The issue was solved by downgrading the Python package **tornado** from 5.1.1 to 4.5.3. On my own development machine for this book project, I have version 5.0.2 installed without issues.

Conditional Logic

The next basic concept that you need to be aware of is how to make

conditional statements. For example, to tell the code that if some value is above another value, then perform some action. Think of a trading logic where you want to buy if the price is above a moving average or similar, which of course we will be doing later in this book.

A conditional statement works in a similar way as the loops we just saw. We use the word `if` to show that we want to make a conditional statement, we need to end that statement with a colon, just as we did for the loops, and we need the indentation for the for following.

Building on the knowledge we just acquired using lists, take a moment to look at the following code.

```
bunch_of_numbers = [
      1, 7, 3, 6, 12, 9, 18
      ]

for number in bunch_of_numbers:
   if number == 3:
      print("The number three!")
   elif number < 10:
      print("{} is below ten.".format(number))
   else:
      print("Number {} is above ten.".format(number))
```

There are a few important points to learn in that brief code segment. The first row should be familiar by now. We're starting by creating a list, this time of a bunch of numbers. After that, we start a loop, iterating through all those numbers, just like we did earlier in this chapter. Nothing new so far.

But then there is that first conditional statement. Take note of the double equal signs. That's not a typo. In Python, just like in most other programming languages, a single equal sign is used to assign a value, while double equal signs are used for comparison. In this case, we are saying "if the current number is equal to three, then do the following".

Also take note here of the nested indentations. After the loop is initiated, there is an indentation for the next block of code. After that, we have the `if` statement, which requires its own indentation to show which code block should be run if the condition is met.

After the `if` statement, you see a like

which may at first not be so obvious. A new strange word, called eli f . This is just short for “else if”. If the number was 3, the first condition was met and we will not reach the eli f row. This is only evaluated if the number was not three, and we did not meet the first condition.

The eli f statement checks if the number is below ten. If so, print out a line of text, and move to next number.

Finally, if neither of the conditions were met, the els e condition will be met. In this case, we know that the number was not 3 and that it was above 10.

Common Mistakes

At this point, it’s probably a good idea to bring up some of the most common sources of errors early on. When first starting out with Python, I recall that the thing that kept getting me snagged was the indentation logic.

Remember that Python groups blocks of code based on how far from the left edge they are. Most of the time, this

indentation is done automatically by the editor. When you end a row with a colon, for example, most Python editors will automatically start you off one tab further to the left when you press enter. It knows that you are about to make a new code block.

But that doesn't prevent the occasional error. If you for instance leave an accidental space at the start of a row, your code will fail to run. The code below is the same as we used earlier to demonstrate conditional statements, but there is a deliberate error in it. See if you can spot it.

```
bunch_of_numbers = [
      1, 7, 3, 6, 12, 9, 18
      ]

for number in bunch_of_numbers:
   if number == 3:
      print("The number three!")
    elif number < 10:
      print("{} is below ten.".format(number))
   else:
      print("Number {} is above ten.".format(number))
```

It's not that easy to see in the code segment above. The row starting with eli f

now has an additional space in front of it, making it misaligned with the if and the else statement. If you try to run this code, you will get a message about "IndendationErro r". You can see this demonstrated in Figure 5-7. This book is in black and white, but if you try this in a notebook, you will see that the word eli f in the code will be automatically highlighted in red, to show that something is wrong here. Look closely, if you don't spot it right away. The row with eli f is not aligned with the if statement above it.

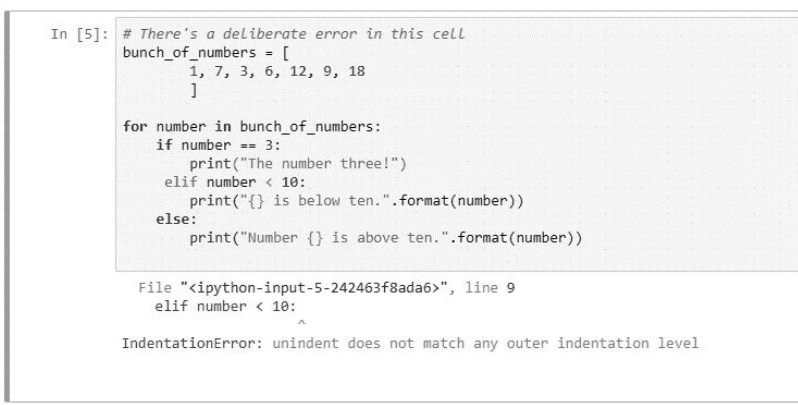

In [5]:
# There's a deliberate error in this cell
bunch_of_numbers = [
1, 7, 3, 6, 12, 9, 18
]
for number in bunch_of_numbers:
if number == 3:
print("The number three!")
elif number < 10:
print("{} is below ten.".format(number))
else:
print("Number {} is above ten.".format(number))
File "<ipython-input-5-242463f8ada6>", line 9
elif number < 10:
^
IndentationError: unindent does not match any outer indentation level

Figure 5-7 Indentation Error

Sometimes Python error messages are friendly and helpful, and sometimes they can be rude and confusing. A good example of a nice and friendly error

message is when you forget the parenthesis for a print statement. That's a very common mistake, as previous versions of Python did not require such a parentheses.

In Figure 5-8 you can see the same code again, with missing parentheses on row 7. In the console you can see how the error message figured out what you did wrong and suggests a fix.

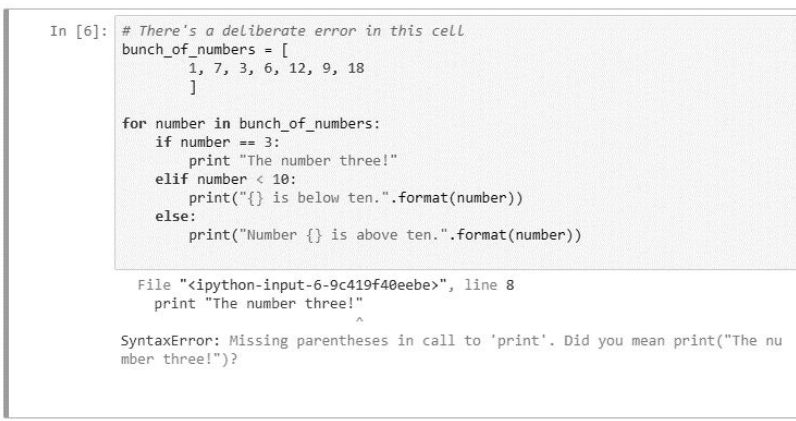

In [6]:
# There's a deliberate error in this cell
bunch_of_numbers = [
1, 7, 3, 6, 12, 9, 18
]
for number in bunch_of_numbers:
if number == 3:
print "The number three!"
elif number < 10:
print("{} is below ten.".format(number))
else:
print("Number {} is above ten.".format(number))
File "<ipython-input-6-9c419f40eebe>", line 8
print "The number three!"
^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("The nu
mber three!")?

Figure 5-8 Missing Parentheses

Installing Libraries

Installation of Python libraries is usually done at the terminal. The terminal can run commands and programs for you,

just like the good old command prompt. In fact, it would be possible to do this in the normal command prompt, but the way I will show you here is easier.

At this point, I merely mention the command line installation as you're likely to run into this when reading other books or online articles on the topic. While it is quite common to use text commands to deal with installations, you can avoid this additional hassle for now.

I'm guessing it's a fair assumption that most readers of this particular book would prefer to avoid text commands wherever possible, so instead I will show you a visual way of accomplishing the same task.

When you installed the **Anaconda** package, as discussed earlier in this chapter, one of the applications installed was **Anaconda Navigator** . Find it, and open it. This application can help simplify a lot of common Python tasks, and you will likely be using it quite a lot.

When you open **Anaconda Navigator** , you should see a sidebar

menu on the left. One of these menu items is Environments. Go there. We will go into environments a bit more later on, as we will soon create a new one. The ability to have multiple environments is great, as we can install environments with specific libraries and versions for specific purposes.

For now, if you just made a fresh installation of **Anaconda** , you will probably only have one item here, which is likely called root or base.

Select the root environment, and the right side of the screen will update to show you exactly which libraries are installed in this environment. As you can see in Figure 5-9, there is also a dropdown where you can select if you want to see libraries that are already installed or those available but not yet installed.

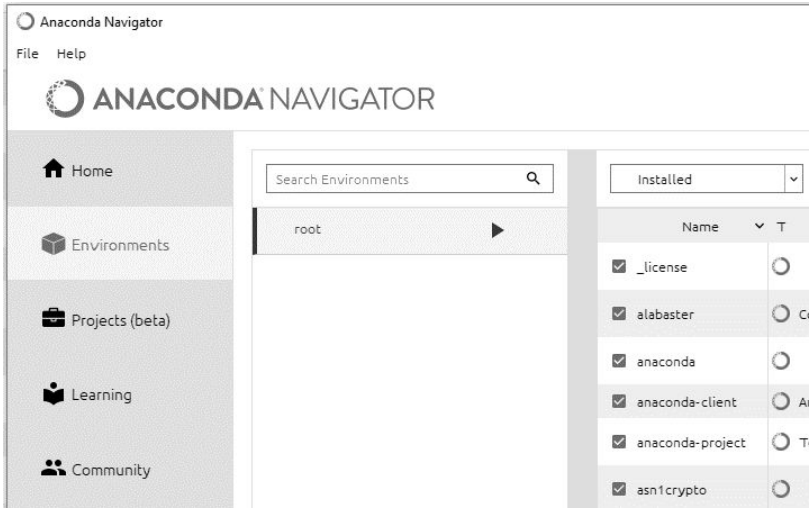


Figure 5-9 Anaconda Libraries

The simplest way to install a new Python library is to go here and select Not Installed in that dropdown, so that you see libraries available for installation. Find the library that you need, check it, and hit the Apply button at the bottom. That's it.

At this point, you're wondering what libraries you need to install, and why. Well, I mention how to install libraries early on, because from time to time you will run into code samples which use libraries which you don't yet have installed. When you do need it, it's better to know in advance what to do.

We will need some quite common libraries early on in this book, so before moving on to next chapter, make sure that you've got them installed. Depending on your installation, you may already have them installed, but let's check.

Select to view all libraries in that dropdown, the one that you can see in Figure 5-9, and then find **Pandas** . You can use the search field on top to find it faster. If there is a check mark next to this library, it's already installed and you're all good. Next verify that **MatPlotLib** is installed, the same way. It if's not installed, click the check box next to it and apply changes.

These are two very common libraries which we will use in practically all sample code from here on in this book. The first library revolutionizes time series processing and is the number one reason for the rise of Python in financial modeling, and the second library is for visualizing data.

It's important to remember that if you are using multiple environments,

which we will do soon in this book, the library you just installed did not install into all of them. Just the one that you had selected, in this case the root or base environment.

Some more complex libraries may be dependent on other libraries. In that case, they will automatically make sure that all dependencies are installed as well. Most backtesting libraries, for instance, will require a set of other libraries to be installed. When we install those backtesting libraries, they will take care of installing all these other required libraries for us.

6 Bring out the Pandas

I told you before that I'm going to drop you right into the deep end. It's more interesting to learn if you see some real use for your knowledge early on. So let's move right along here, and get something useful done.

The **Pandas** library is an absolute game changer in the world of Python. There are entire books written just about **Pandas** , most notably the book Python for *Data Analysis* , by the guy who actually wrote the **Pandas** library to begin with, Wes McKinney (McKinney, 2017). I won't go into anywhere near the kind of details that he does, but I will keep using his brilliant library over and over in this book.

The **Pandas** , commonly spelled in all lower caps for reasons that I haven't fully understood and therefore refuse to comply with, is a library to handle structured data. Most importantly for us traders, it excels at dealing with time series data.

For our next trick, we will read time series data from a file, calculate a moving average on this and then show a graph. That sounds a little more complex than the loop stuff you did before, right?

The first question is, where we get the data from. If you pick up any random Python book written before 2017, they

probably show you how to automatically get data from Yahoo or Google in a single line of code. Well, as it turns out, that did not work out very well, no fault of the authors. Both Yahoo and Google decided to shut down these services without prior notice, leaving thousands of code samples in books and websites permanently bricked.

For this first exercise, I will assume that you have a local comma separated file with some data that we can use. If you don't have one handy, download one from my website **www.followingthetrend.com/trading-evolved** . Keeping our first time series experiment simple, the layout of your file has two columns only. The first one with dates and the second with prices. Here is how my file looks.

```
Date,SP500
2009-06-23,895.1
2009-06-24,900.94
2009-06-25,920.26
2009-06-26,918.9
2009-06-29,927.23
2009-06-30,919.32
```

```
2009-07-01,923.33
2009-07-02,896.42
2009-07-06,898.72
2009-07-07,881.03
2009-07-08,879.56
2009-07-09,882.68
2009-07-10,879.13
2009-07-13,901.05
2009-07-14,905.84
2009-07-15,932.68
2009-07-16,940.74
2009-07-17,940.38
2009-07-20,951.13
...
```

In case you reside on the other side of the Pond from me, you may take issue with my choice of date format. I use `yyyy-mm-d d` format, which is common around my parts but might not be where you live. No worries. It does not matter. Use whatever date format you are comfortable with. Pandas will figure it out for you later.

For convenience, place this csv file in the same folder where you will save the Python code. You can put it anywhere you like of course, but if it's not in the same folder you will have to specify a path in

the code, so that we can find the file.

Now we are ready to build the code. Get back to **Jupyter Notebook** and make a new file. This time, we will learn a few more new concepts. The first one is to import a library to use with our code, in this case the **Pandas** .

To read this data, calculate a moving average and show a chart, all we have to do is the following code.

```
%matplotlib inline
import pandas as pd
data = pd.read_csv('sp500.csv', index_col='Date',
parse_dates=['Date'])
data['SMA'] = data['SP500'].rolling(50).mean()
data.plot()
```

That's really not bad, is it? Consider how easy that was, compared to what it would take in most other programming environments.

Here is what this code does. The first quirky looking row, %matplotlib inlin e , is something you will see a lot from now on. The details of why this row is needed are not important at the moment, but this row is required to make sure that graphs will be shown in the notebook. If you

forget that row, you will only get text output and no graph.

Next, we tell our code that we want to use the **Pandas** library, `import pandas as pd`. As is common, we make an alias for **Pandas**, so that we can refer to it by `pd`, instead of `panda s`. That's just to avoid having to type the whole word over and over again, as this library is likely to be used a lot going forward. You will see this alias often, and whenever code in this book refers to `pd`, that would be a reference to the **Pandas** library.

After this we see the following row: `data = pd.read_csv('sp500.csv', index_col='Date', parse_dates=['Date'])`, reads the file from disk into the variable called `data`. We specify here that the column with the header `Date` is the index, and that we want **Pandas** to parse the date format of this column. See, I told you that your choice of date format does not matter. Even though the European one is clearly superior. This is where **Pandas** will sort it out for you.

Then we have the next row, `data['SMA'] = data['SP500'].rolling(50).mean()`,

which adds a moving average, or as it's usually referred to by fancy data scientists, a rolling mean. As you may already surmise, there are a plethora of other functions that can be applied on a rolling window this way, using the same syntax.

The rolling mean here is calculated on the column `sp50 0` . In case you are wondering where that name comes from, the answer is simple. Look at the layout of my sample csv file again. That was the name of the second column, the one containing the index closing prices.

Pandas will read the names of the headers, and you can refer to them later in this manner. The object that we created by reading the csv file is called a **DataFrame** . That's an important **Pandas** concept that we will return to many times in this book.

Think of a **DataFrame** as a spreadsheet. Just better. It has rows and columns, just like a spreadsheet, and you can easily perform mathematical functions on it. A **DataFrame** has an index, which

can be either just row numbers or something more useful such as dates in the case of time series. Columns can have named headers, making it easy to refer to them.

And the final row of our code? It creates a chart of our data, using the simple function call plot( ) . The output should look pretty much like Figure 6-1.

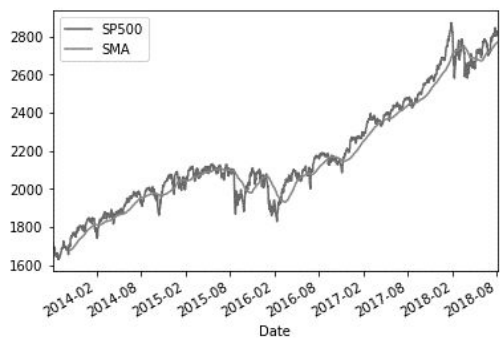


Figure 6-1 Our first Chart

That's pretty neat, right? We pulled in data, parsed dates, calculated analytics and built a chart, and all in just a few lines of code. I hope at this point, if you are new to all of this, that the value of Python should start to become clearer.

The really interesting row here is really the one that calculates the rolling mean. Obviously the math is dead simple,

but that's beside the point. The really exciting thing here is how it's done. That we can simply take a time series, get a rolling window and do math on this directly.

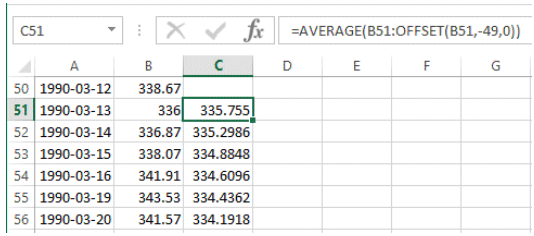
To understand why this is so neat, take a moment to consider how you would have done the same thing in Excel.

Opening the csv file in Excel is no issue of course. You would then scroll down to the 50 th data point, probably on row 51 since you have column headers at top. There, you would write a formula like =AVERAGE(B51:OFFSET(B51,-49,0)) . Then you would need to copy this all the way down. This means that you will have a large number of individual functions in your sheet already. And don't forget that Excel keeps recalculating every single formula, any time you change anything in the spreadsheet. That being one of major issues with Excel of course.

The offset in Excel would need to be 49 and not 50, as the starting cell, B51, is counted as well.

Using Python, we can apply a

function on an entire time series at once. In this case, it's simple math, but as you will see later, it works the same way with complex calculations.



	A	B	C	D	E	F	G
50	1990-03-12	338.67					
51	1990-03-13	336	335.755				
52	1990-03-14	336.87	335.2986				
53	1990-03-15	338.07	334.8848				
54	1990-03-16	341.91	334.6096				
55	1990-03-19	343.53	334.4362				
56	1990-03-20	341.57	334.1918				

Figure 6-2 Moving Average in Excel

With Excel, this simple task requires thousands of individual formulas and the mixing of data and logic in the same file. Now imagine if we want to shift between many different financial time series, and many different analytics. The Excel file would grow increasingly complex and would quickly get unmaintainable. The Python way is far superior.

Documentation and Help

After seeing that code sample in the previous section, it would be fair to ask

how you could possibly know what argument to use in order to get Pandas to set an index column, or to parse the dates. Not to mention how you could know what other possible arguments there are.

You could approach this in two ways, which will probably yield the same information in the end. One way would be to search the internet, which will likely give you both the official documentation and various usage samples in the first few returns from your search engine of choice.

Then again, you might as well just look up the details in the built-in documentation. It's all there, and will pop up on your screen if you say the magic words. All, or at least the vast majority of Python libraries have this kind of built in documentation. As a demonstration, let's use **Pandas** to find out about what it is, how it works, what functions are available and finally how exactly to use `read_csv()`.

Open up a new Jupyter Notebook again. Or if you're feeling really lazy, download the sample from the book website, as all code samples are available

there. First import Pandas, the same way as we've done before.

```
import pandas as pd
```

Now we can refer to is by the alias `pd`. In the next cell, simply run this line of code.

```
help(pd)
```

That will show you an overview of what **Pandas** is, which version you're using and some technical information that you're probably not too interested in at the moment. Your output should look more or less like the text below.

```
Help on package pandas:

NAME
    pandas

DESCRIPTION
    pandas - a powerful data analysis and
manipulation library for Python

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =


    **pandas** is a Python package providing fast,
```

flexible, and expressive data
    structures designed to make working with "relational" or "labeled" data both
    easy and intuitive. It aims to be the fundamental high-level building block for
    doing practical, **real world** data analysis in Python. Additionally, it has
    the broader goal of becoming **the most powerful and flexible open source data
    analysis / manipulation tool available in any language**. It is already well on
    its way toward this goal.

    Main Features

    Here are just a few of the things that pandas does well:

      - Easy handling of missing data in floating point as well as non-floating
        point data
      - Size mutability: columns can be inserted and deleted from DataFrame and
        higher dimensional objects
      - Automatic and explicit data alignment: objects can be explicitly aligned
        to a set of labels, or the user can simply ignore the labels and let
        `Series`, `DataFrame`, etc. automatically

align the data for you in
computations
- Powerful, flexible group by functionality to perform split-apply-combine
operations on data sets, for both aggregating and transforming data
- Make it easy to convert ragged, differently-indexed data in other Python
and NumPy data structures into DataFrame objects
- Intelligent label-based slicing, fancy indexing, and subsetting of large
data sets
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from flat files (CSV and delimited),
Excel files, databases, and saving/loading data from the ultrafast HDF5
format
- Time series-specific functionality: date range generation and frequency
conversion, moving window statistics, moving window linear regressions,
date shifting and lagging, etc.
...

The help function gives us some overview information about the **Pandas**

library, but it lacks the details about what objects are in there, and how they work. No worries, we can go deeper in the documentation.

We can run the same help function on the **DataFrame** as well. Try executing the following line in the **Jupyter Notebook** .

```
help(pd.DataFrame)
```

That will give you quite a long list of functionality built into the **DataFrame** object. As general rule, for now you can safely ignore all the built in functions starting with underscores. The text that comes up tells you what a **DataFrame** is, what purpose it has, and it lists functions and features.

You can take this help business a step further and ask for the details on the **read_csv()** function itself.

```
help(pd.read_csv)
```

Executing that line will show you all possible arguments that can be used for reading a csv file, what the default values of these are as well as a description

of each argument. That should tell you all that you need to know about how to use this function.

In the example here, we were looking for information on the index column and the date parsing. So let's look closer at what this documentation says about those.

index_col : int or sequence or False, default None
Column to use as the row labels of the DataFrame. If a sequence is given, a
MultiIndex is used. If you have a malformed file with delimiters at the end
of each line, you might consider index_col = False to force pandas to _not_
use the first column as the index (row names)

This tells us that we could decide on index columns by providing a number or a name, but we can also decide not to have a number. If you use a number, remember that everything in Python world is zero based, so the first column in your file would have number 0.

Then we'll check the same documentation for parsing dates. That

expression refers to having the code analyze the text string and figure out how to make a date out of it.

parse_dates : boolean or list of ints or names or list of lists or dict, default False

* boolean. If True -> try parsing the index.
* list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3
each as a separate date column.
* list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as
a single date column.
* dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result
'foo'

If a column or index contains an unparseable date, the entire column or
index will be returned unaltered as an object data type. For non-standard
datetime parsing, use ``pd.to_datetime`` after ``pd.read_csv``

This text tells us that if we want to parse dates, one way would be to simply set parse_dates=True. If we do that, **Pandas** will try to make dates out of the index

column. Sometimes we may want to tell it to parse other columns, so we also have the option to specify which columns should be analyzed as dates and parsed, by providing a list of column numbers or names.

This way of obtaining documentation can be used for practically anything. If you get lost, either type the function name into a search engine, or use the built in help functionality.

Simple Python Simulation

Now we are going to step this up a bit with another little demo of how quickly Python can get useful. For this demo, we will use the same S&P 500 data and build a simulation. What would happen if we would go long when a 50 day moving average crosses above a 100 day moving average, and close when it crosses back down?

No, I'm not saying that's a good

strategy to trade. This is merely an example, and one which is very easy to code. Also, we are not aiming for realism here. Not in this chapter. This is after all the Python intro chapter. So for now, things are going to be kept fairly simple.

For this simulation, we are going to use the **Pandas** library again, as well as one more very useful and very common library. **Numpy** , short for Numerical Python, is a library with all kinds of useful mathematical functions. They make life in Python Land easier, as well as quicker. Just as **Pandas** is usually aliased as `pd` , **Numpy** is usually aliased as `np` . That's the way I will do it here, and throughout the book.

While **Numpy** is most likely already installed in your root environment, you could always verify in **Anaconda Navigator** , as we looked at in chapter 5. If it's not installed, go ahead and install it. This is another library which you will probably be using quite a lot.

We will do this step by step, to make sure you follow the logic. And trust

me, this is not complex stuff. At least not yet. We are going to about 10 rows of code here, no more. It could be done with less even.

```
# Make sure the plot shows up
%matplotlib inline

# Import libraries that we need
import pandas as pd
import numpy as np
```

Ok, so those initial lines should be clear by now. First that row to make sure that the graphs appear in the notebook, and then a couple of import statements, so that we can use the functionality of **Numpy** and **Pandas** .

```
# Read the data from disk
data = pd.read_csv('sp500.csv', index_col='Date',
parse_dates=['Date'])
```

Well, this one we have seen before. We are reading the data from a file, into a **Pandas DataFrame** .

```
# Calculate two moving averages
data['SMA50'] = data['SP500'].rolling(50).mean()
data['SMA100'] = data['SP500'].rolling(100).mean()
```

These lines of code calculate the two moving averages that we are going to

use for the simulation. As you see, they just reference a rolling time window of 50 and 100 rows respectively, and calculates a mean of those.

```
# Set to 1 if SMA50 is above SMA100
data['Position'] = np.where(data['SMA50'] >
data['SMA100'], 1, 0)
```

The next line checks which days we should be long, and which days we should be flat. The strategy being that we will be long if the faster moving average is above the slower. That is, when the column SMA50 is above SMA100 , we will be long. Else we are going to be flat.

What the line above does is to set the column Position to 1 where SMA50 is higher than SMA100 , and set all other days as 0 .

But, here's an important part in terms of the logic. At this point in the code, the position column changes the same day as the average crosses over. That is, we trade instantly at the closing price, the same value that we base our average calculation on, and thereby our signal. That's clearly cheating. To make

this even reasonably fair, we need to delay the trade to the day after the signal. Luckily, this is very easily done.

```
# Buy a day delayed, shift the column
data['Position'] = data['Position'].shift()
```

Next we calculate how many percent per day the strategy changes. This is easy, if you think about it. We know how many percent the index moves per day. And we know that we will be long 100% of the index if the 50 day moving average is above the 100. If that's not the case, we are not owning any.

```
# Calculate the daily percent returns of strategy
data['StrategyPct'] = data['SP500'].pct_change(1) *
data['Position']
```

We take the daily percent change of the index, and multiply by the Positio n column that we just created. Remember that this column will be 1 if we should be long, else 0.

Notice how easily you can get the percentage return from the index. We just refer to that column in our Pandas **DataFrame** , and call the function pct_change( ) .

```
# Calculate cumulative returns
data['Strategy'] = (data['StrategyPct'] + 1).cumprod()
```

Next it's time to calculate the return of our strategy. We already know the daily return numbers, and we just need to make a time series out of it. The simplest way would be to add the number 1 to all the percentage returns, and then use the Pandas function `cumprod( )` . This will calculate a cumulative product of the series.

Note in the above line of code that we can add a number to each row in the column, simply by using the plus sign. Other programming languages would complain that we are mixing concepts, trying to add a static number to a list of numbers. But Pandas will figure out what we mean, and add the number to every single row.

```
# Calculate index cumulative returns
data['BuyHold'] = (data['SP500'].pct_change(1) +
1).cumprod()
```

It would be useful to plot the buy and hold alternative, next to the strategy, so that we can compare the two. The method here is the same as when we

calculated the strategy returns, we just base it on the actual S&P time series instead.

```
# Plot the result
data[['Strategy', 'BuyHold']].plot()
```

Finally, the last row, plotting the two lines. Last time we plotted something, we did not specify which columns we wanted to plot, because there were only two columns and we wanted to plot them both. If we don't specify which columns to plot, we would get all of them. And that would look like spaghetti. All we care about here is to show the equity curve of our strategy next to the buy and hold benchmark, and that's what this does.

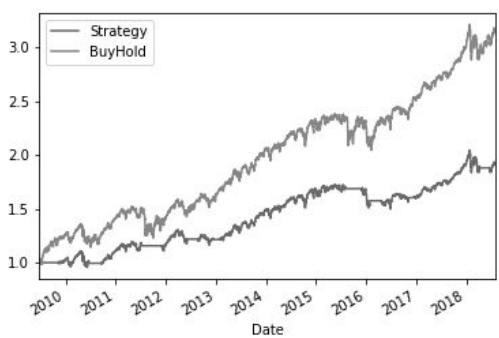


Figure 6-3 Simple Simulation Results

Given the black and white nature of this book, and the fact that we haven't yet

learnt how to make dashed or dotted lines in a Python plot. We'll get to that later. But in case you're wondering that's our strategy at the bottom in the chart. Showing lower returns, but also seemingly lower volatility. Clearly a graph like this is not enough to decide if the strategy is any good, but that's not what we care about at the moment.

You just made your first Python backtest. Congratulations!

Most of you are probably far more familiar with Excel than with **Pandas DataFrames** , and it may be helpful to see how the columns we calculated would have looked in Excel. The **Dataframe** that we created will be structured as in Figure 6-4. The first column, SP500, comes straight from the data file, while all others are calculated. Depending on if the SMA50 is higher than SMA100, the column Positio n will show either 1 or 0. If it shows 1, we assume that the strategy is holding a position of 100% of the portfolio value, and multiply the previous strategy value with the day's percent change in the index.

Date	SP500	SMA50	SMA100	Position	StrategyPct	Strategy	BuyHold
2010-06-03	1,102.83	1,158.66	1,137.63	1	0.41%	1.00	1.23
2010-06-04	1,064.88	1,156.60	1,136.81	1	-3.44%	0.97	1.19
2010-06-07	1,050.47	1,154.29	1,135.95	1	-1.35%	0.96	1.17
2010-06-08	1,062.00	1,152.20	1,135.11	1	1.10%	0.97	1.19
2010-06-09	1,055.69	1,149.85	1,134.18	1	-0.59%	0.96	1.18
2010-06-10	1,086.84	1,148.12	1,133.69	1	2.95%	0.99	1.21
2010-06-11	1,091.60	1,146.57	1,133.11	1	0.44%	0.99	1.22
2010-06-14	1,089.63	1,144.80	1,132.62	1	-0.18%	0.99	1.22
2010-06-15	1,115.23	1,143.35	1,132.61	1	2.35%	1.02	1.25
2010-06-16	1,114.61	1,141.86	1,132.84	1	-0.06%	1.01	1.25
2010-06-17	1,116.04	1,140.53	1,133.03	1	0.13%	1.02	1.25
2010-06-18	1,117.51	1,139.15	1,133.28	1	0.13%	1.02	1.25
2010-06-21	1,113.20	1,137.53	1,133.44	1	-0.39%	1.01	1.24
2010-06-22	1,095.31	1,135.50	1,133.55	1	-1.61%	1.00	1.22
2010-06-23	1,092.04	1,133.40	1,133.73	1	-0.30%	0.99	1.22
2010-06-24	1,073.69	1,130.66	1,133.58	0	0.00%	0.99	1.20
2010-06-25	1,076.76	1,127.96	1,133.31	0	0.00%	0.99	1.20
2010-06-28	1,074.57	1,125.61	1,133.08	0	0.00%	0.99	1.20
2010-06-29	1,041.24	1,122.48	1,132.86	0	0.00%	0.99	1.16
2010-06-30	1,030.71	1,118.95	1,132.51	0	0.00%	0.99	1.15
2010-07-01	1,027.37	1,115.38	1,132.22	0	0.00%	0.99	1.15
2010-07-02	1,022.58	1,111.66	1,131.74	0	0.00%	0.99	1.14
2010-07-06	1,028.06	1,107.88	1,131.34	0	0.00%	0.99	1.15
2010-07-07	1,060.27	1,104.84	1,131.15	0	0.00%	0.99	1.18
2010-07-08	1,070.25	1,102.57	1,131.10	0	0.00%	0.99	1.20
2010-07-09	1,077.96	1,100.30	1,130.93	0	0.00%	0.99	1.20

Figure 6-4 Excel Table View of Simple Simulation Data

If you followed this example so far, you are hopefully getting a bit impressed over how Python can get things done really quickly and easily. Then again, perhaps you are already starting to pick this example apart and question the validity of this simulation.

Some would now question if this really a legitimate simulation, given how it does not account for transactions costs, for example. Well, if that was your concern regarding validity, you missed the

big one. We are 'trading' the S&P 500 Index here. That's not a tradeable security. You can't trade an index. And that's not the only issue.

So no, this simulation we just did, is hardly built for realism. There are plenty of issues with it, and there are good reasons for why we don't just do all our simulations with such simple logic. But that was of course not the point here. The point is to demonstrate some neat things with the language Python.

Before moving on to realistic simulations, we are going to look at a few more easy and fun examples to get everyone up to speed on Python and the neat things it can do for us.

Making a Correlation Graph

There are a couple of more common concepts that I would like to demonstrate to you before moving on. Open up a new file in **Jupyter Notebook** . This time we

are going to make a graph showing correlation over time between the S&P 500 and the Nasdaq.

The most important point that I want to demonstrate this time is how functions work in Python. We are going to make a flexible and reusable function for fetching data from disk. This function can then be used to read the different csv files. Once we have that data, we will use that data to calculate rolling correlations, before plotting the result. Easy.

When it comes to calculating correlations, you will likely notice that most practitioners prefer to use log returns. The reason for that is that it can be very convenient to work with log returns when processing and analyzing time series data, but it has no discernable impact on the end result.

The point here is to demonstrate concepts, and for now I will stick to good old percentage returns. Nothing wrong with that. What's important to understand however is that for correlations to make sense, you do need to use one of these two

alternatives. Either log returns or percentage returns. Calculating correlations on the price levels themselves wouldn't make logical sense, and neither would using the dollar change.

Here is the code for our correlation plot program.

```
%matplotlib inline
import pandas as pd

def get_returns(file):
    """
    This function get_data reads a data file from disk
    and returns percentage returns.
    """
    return pd.read_csv(file + '.csv', index_col=0,
parse_dates=True).pct_change()

# Get the S&P time series from disk
df = get_returns('SP500')

# Add a column for the Nasdaq
df['NDX'] = get_returns('NDX')

# Calculate correlations, plot the last 200 data points.
df['SP500'].rolling(50).corr(df['NDX'])[-200:].plot()
```

Notice how in this code, there are comments that help explain what is going on. It's a good idea to write comments in the code, both for yourself to remember

things and for others to be able to read it.

As you see here, there are two ways of making comments. One is a block comment, which is bookended by triple quotation marks. Anything written between the triple quotes will be deemed a comment.

The second way is to use a hash sign, and everything after that character will become a comment.

To the actual code. See that row starting with the keyword de f ? That defines a function. A function is a piece of code, encapsulating some functionality which hopefully can be reused. This simple function, get_return s , takes a file name as an argument. We give that function a file name, and it will read that file from disk, if it exists in the same folder as the program, and returns daily percent returns. We have seen this before.

We are going to call this function twice. First to read the S&P 500 data from disk into a **Pandas DataFrame** , and then again to add the Nasdaq data to that **DataFrame** .

When you try this out, you could either use your own data for this, or download sample files at **www.followingthetrend.com/trading-evolved** .

This is the layout that I used for my files:

```
Date,SP500
1990-01-02,359.69
1990-01-03,358.76
1990-01-04,355.67
1990-01-05,352.2
1990-01-08,353.79
1990-01-09,349.62
1990-01-10,347.31
1990-01-11,348.53
1990-01-12,339.93
1990-01-15,337
1990-01-16,340.75
1990-01-17,337.4
1990-01-18,338.19
...
```

After this, it gets interesting. This row is meant to teach you something about **Pandas** , and how easily you can do calculations on it.

Once we have the **DataFrame** with

the two time series, we use a single line of code for multiple, complex operations. In a single row, the last row of that code sample, we do the following:

- Apply a correlation formula on a rolling 50 day window of the percent returns for the indexes.
- Discard all data except the last 200 rows.
- Draw a plot of the result.

Now imagine doing the same thing in Microsoft Excel. You would need a very large number of formulas. One for each individual calculation. But with Python, you just got the task done in a single line.

The function here, `get_returns(file )` , is not executed until some other code calls it. Even though the code is run in order, from the top down, the function is skipped over, until some other code requests it.

As you see we can call the same function many times, using different arguments to read different files, instead of writing this code over and over every time. That cuts down on the amount of code we need and makes it easier to maintain and find errors.

Now, after calling the function twice, once for each file that we are reading, the variable `df` holds a **DataFrame** with our daily percent return data. The file would look more or less like Table 6.1. That's an important point to keep in mind here, that a **DataFrame** can be thought of as a table, or a spreadsheet if you like.

Table 6.1 Daily Percent Returns Data

Date	SP500		
-0.0	[illegible]	[illegible]	32
-0.0	[illegible]	[illegible]	21
-0.0	[illegible]	[illegible]	94
0.0	[illegible]	[illegible]	01
-0.0	[illegible]	[illegible]	74
-0.0	[illegible]	[illegible]	64

The source data files, `SP500.csv` and `NDX.csv`, contain in my case data from 1990 onwards. That's a lot of data points, and we only want to plot the last 200 rows here. This is daily data, so one row per day.

We will do this by using a powerful concept called slicing. We are going to

slice the data. With this trick, we can refer to part of a **DataFrame** , and the same logic works for many other types of objects as well.

The basics are very simple. You can slice an object by using the syntax [start:stop:step] after the object. This is important, so we will take a moment to look at that. Slicing is something that you really want to understand.

For a sequence of data, such as a **DataFrame** , we can select a part of it using this concept. If we wanted to just refer to just a part of the object refs that we just created, or any similar object, we could use this slicing.

If we have an object called data and wanted to refer to a segment starting at the 100 th row and ending at the 200 th row, taking every second row between those, we would use data[100:200:2] . Simple and straight forward. We don't have to supply all three, if we don't want to. Leave out the step number, and you will get all the rows.

The syntax data[50:60] will give you

all rows from row 50 to row 60. If you write `data[-50:-20 ]` you will get rows starting from the 50 th from the end, to the 20 th from the end. And if you simply say `data[-10: ]` you will get data from the tenth last, to the very last row.

In his example, I want to plot the last 200 points only, simply to make the graph easy to read. We can do this by using a negative value to refer to the starting point, which **Pandas** will interpret as number of points from the end, instead of from the start.

Remember that the variable we have created in the example is called `d f` , a completely arbitrary name of course. So if we type `df[-200: ]` that would refer to the last 200 points of the `d f` object. We need that colon sign in there, to make it clear that we are going for a slice. If you would simply write `df[200 ]` we would just get one single row, the 200 th row.

But we are not plotting the `d f` objects itself. We want to plot the correlations based on the data in `d f` .

Look at that last row in the code

sample, `df['SP500'].rolling(50).corr(df['NDX'])[-200:].plot( )` . It actually does multiple things in one line. It starts with the column 'SP500 ' , references a rolling window of 50 rows. It then calculates the correlation against column 'NDX ' . That's a very easy way to make a time series of correlations.

Next you see the brackets, which is where we slice the data. We start at row -200, until the end of the series. Finally, we plot this series. And the output should look like Figure 6-5.

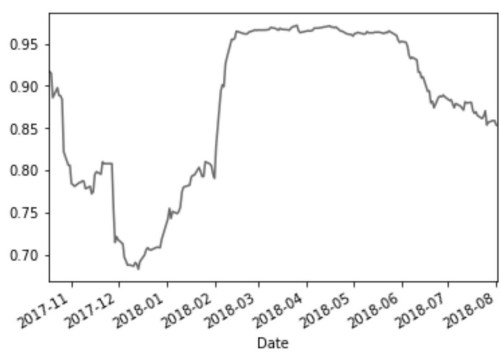


Figure 6-5 Correlation Graph

I hope by this point that you are starting to see how useful Python can be for working with financial data. You should start to get a hang of the basics on how to deal with time series, and as this

book progresses, I will introduce more concepts, tools and tricks for you to use.

Regardless of what technical knowledge you started this book with, my intention is that by the end of it, you should be able to feel comfortable with financial modelling in Python and be able to use your own creativity and solve the tasks that matter to you.

Perhaps most importantly, you should be able to construct realistic trading backtests to test and validate your own trading ideas.

Prettier Graphs

We have made a few simple graphs so far, and we have seen just how simple that can be. Once we have our data in a neat **DataFrame** , we simply tag a .plot( ) on the end, and Bob's your uncle.

That way of making a visual representation of something is really useful for such ad-hoc calculations that you just want to take a quick look at. But it's really quite rudimentary. Often you

may want to make more complex and nicer looking graphs, and you want to be able to control every aspect of the presentation.

In the previous section of this chapter, we installed **matplotlib** , and that's what we are going to use here. With this library, you can make very detailed and very complex plots, if that's your inclination. Sometimes when working with Python, all you want is a quick graphical output to eyeball. Other times, you want something that can impress in presentation material or something very clear for colleagues to be able to interpret and use.

This is still an introductory chapter, and we are going to keep things relatively simple. We are going to use the same data as for the correlation example above. This time, we will plot three subplots, or chart panes if you like. The first will show a rebased comparison of the two indexes, recalculated with the same starting value of 1. This will be shown on a semi logarithmic scale.

The second subplot will show relative strength of the Nasdaq to the S&P 500, and the third the correlation between the two. Mathematically very simple stuff.

As with all code examples in this book, the intention is to keep it easy to read and understand, not to make the most efficient or pretty code. There is a value in making highly efficient and pretty code, but first you need to understand the basics.

I will show you the code bit by bit and explain what's important, and by the end of this section I will show you the complete code all together.

In the code below, we are using similar techniques as previous examples to calculate the data that we need. There are two functions in the code, both of which should at this point seem familiar and understandable.

```
def get_data(file):
    """
    Fetch data from disk
    """
    data = pd.read_csv(file + '.csv', index_col='Date',
parse_dates=['Date'])
```

```
    return data

def calc_corr(ser1, ser2, window):
    """
    Calculates correlation between two series.
    """
    ret1 = ser1.pct_change()
    ret2 = ser2.pct_change()
    corr = ret1.rolling(window).corr(ret2)
    return corr
```

The first function, get_dat a simply reads the data from file and hands back to us. The second, calc_cor r takes two time series as input as well as a time window, and then calculates and returns the resulting correlation series.

Then we set a number of data points that we are planning to plot. If you change this value, the output plot will change accordingly.

```
# Define how many points we intend to plot. Points in
this case would be trading days.
points_to_plot = 300

# Go get the log return data.
data = get_data('indexes')
```

Look at the part below where we rebase the data. Rebasing is about making the data series start at the same initial

value, so that they can be visually compared when we plot them. There are some neat tricks there which I wanted to demonstrate.

```
# Rebase the two series to the same point in time,
starting where the plot will start.
for ind in data:
   data[ind + '_rebased'] = (data[-points_to_plot:]
[ind].pct_change() + 1).cumprod()
```

In this code, we loop through each column in the **DataFrame** . For each of them, we create a new column for the rebased value. What we want to achieve here is to have the two series start with the same value, so that they can be compared visually in a graph. Usually when rebasing, you would recalculate all series to start at either 1 or 100.

In this example, we only want to plot the last 300 points, as we defined earlier in the code. The point of rebalancing is to make it look nice in the graph and make it easy to visually compare. Therefore, we recalculate everything to start at 300 points from the end, all starting with a value of 1.

Note how we used the same slicing

concept as previously to select the data. In this code, we use the same syntax to slice off the last 300 points, and then get the percent return of one column at a time, and add the number one. When we add that number, it's added to every single row. So after that operation, every row contains the daily percentage chance, plus one. But the row is not even over yet.

Now on that we run `cumprod( )` , calculating the cumulative product, which gives us a rebased series starting at 1. All of this in a single, simple row of code.

Look at that row of code again. Reading the text describing it may seem complicated, but the code is amazingly simple. Think of it as an Excel spreadsheet. We take one column at a time. Calculate percent change, row by row. Add one to that, and calculate a running cumulative product. That is, we start from the beginning and multiply these rows by each other, all the way down.

After this operation, the **DataFrame** data should look more or less as in Table

6.2. We added two new columns, with the rebased values.

Table 6.2 DataFrame Contents

Calculating the relative strength is easy, just one series divided by the other. The correlation calculation should also be very familiar by now.

```
# Relative strength, NDX to SP500
data['rel_str'] = data['NDX'] / data['SP500']

# Calculate 50 day rolling correlation
data['corr'] = calc_corr(data['NDX'], data['SP500'],
100)
```

Now all the data we need is there in our **DataFrame** , calculated and ready to be displayed. At this point, you might want to stop and inspect the data, to make sure it looks as you expected. And of course, there are some useful little tools for that.

In the code sample file that you can download from the book website, I have divided up the code into different cells to make it easy to manage. The code you saw so far in this section is in one cell, which fetches and calculates the data without visualizing it.

If you, after having executed this first cell, make a new cell below, you can stop and take a look at how the DataFram e looks.

One way would be to copy the entire **DataFrame** into clipboard, so that you could paste it into Excel and take a

closer look.

```
# You could use this copy the DataFrame to clipboard,
# which could easily be pasted into Excel or similar
# for inspection.
data.to_clipboard()
```

But perhaps you just want to make sure that the layout is as it should and that the values seem reasonable. You could use head( ) or tail( ) to view just the first or the last part of the **DataFrame** .

```
# We can take a look at the data in the DataFrame
# Using head or tail to print from the start or the
bottom.
data.tail(20)
```

If you're happy with how the **DataFrame** looks now, we'll move on to slicing the data, effectively discarding all but the last 300 data points, using the slicing logic from before.

```
# Slice the data, cut points we don't intend to plot.
plot_data = data[-points_to_plot:]
```

Now we have all the data ready, and all we need to do is to create a nice looking graph. As you can see here, we have more freedom to format and configure the graph, compared to just the simple .plot( ) which we used before.

In this case, we start by saying that we would like a larger figure, defining the exact size.

```
# Make new figure and set the size.
fig = plt.figure(figsize=(12, 8))
```

Another important point here is the subplot function, where we give three values. As you see here, the first time we call this function, we give the numbers 311. That means that we intend to build a figure which is three sub plots high, and just one wide. The final number 1 states that we are now defining the first of these three sub plots.

For the first chart pane, or figure as it's called here, as opposed to the entire plot, we first set a title, and then add two semi-log plots. Note how we set one to be solid line and the other dashed, using linestyle, as well as setting labels and line widths.

```
# The first subplot, planning for 3 plots high, 1 plot
wide, this being the first.
ax = fig.add_subplot(311)
ax.set_title('Index Comparison')
ax.semilogy(plot_data['SP500_rebased'], linestyle='-',
label='S&P 500', linewidth=3.0)
ax.semilogy(plot_data['NDX_rebased'], linestyle='--',
```

```
label = 'Nasdaq', linewidth = 3.0)
ax.legend()
ax.grid(False)
```

To make the second figure, we call add_subplot( ) again, this time providing the numbers 312, showing that we are working on a plot that's three figures high and one figure wide, and this is the second one.

The second time we call it, we give the numbers 312, declaring that this time we intend to define how the second subplot should look. Here we add the relative strength, and add a label for it. Lastly, we add the third figure and we're all set.

```
# Second sub plot.
ax = fig.add_subplot(312)
ax.plot(plot_data['rel_str'], label = 'Relative Strength,
Nasdaq to S&P 500', linestyle = ':', linewidth = 3.0)
ax.legend()
ax.grid(True)

# Third subplot.
ax = fig.add_subplot(313)
ax.plot(plot_data['corr'], label = 'Correlation between
Nasdaq and S&P 500', linestyle = '-.', linewidth = 3.0)
ax.legend()
ax.grid(True)
```

Also note in the code that we add legends to the figures, as well as add gridlines to the last two. Again, I added those things just to show you how.

The output of this code should show a figure with three sub plots, each under the next. We have defined names for each line and a name for the overall figure. The first subplot is set to semi logarithmic axes, and we want a grid on the last two subplots. When running this, you should see something very similar to Figure 6-6.

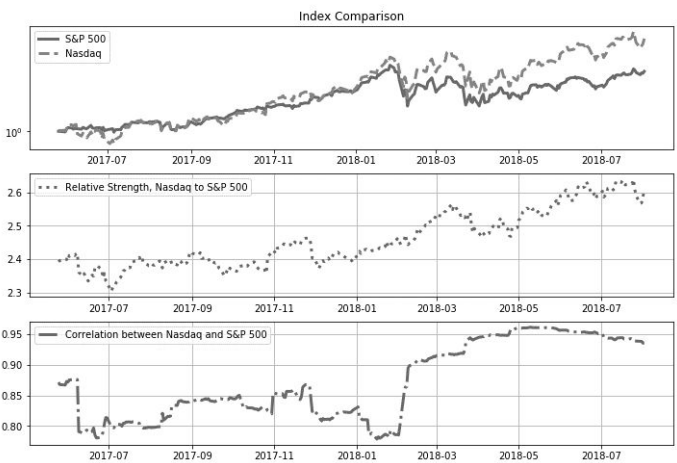


Figure 6-6 Prettier Graph

The full code in one go would look like this right below.

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
def get_data(file):
    """
    Fetch data from disk
    """
    data = pd.read_csv(file + '.csv', index_col='Date',
parse_dates=['Date'])
    return data
```

```
def calc_corr(ser1, ser2, window):
    """
    Calculates correlation between two series.
    """
    ret1 = ser1.pct_change()
    ret2 = ser2.pct_change()
    corr = ret1.rolling(window).corr(ret2)
    return corr

# Define how many points we intend to plot. Points in
this case would be trading days.
points_to_plot = 300

# Go get the data.
data = get_data('indexes')

# Rebase the two series to the same point in time,
starting where the plot will start.
for ind in data:
    data[ind + '_rebased'] = (data[-points_to_plot:]
[ind].pct_change() + 1).cumprod()

# Relative strength, NDX to SP500
data['rel_str'] = data['NDX'] / data['SP500']
```

```
# Calculate 50 day rolling correlation
data['corr'] = calc_corr(data['NDX'], data['SP500'],
100)

# Slice the data, cut points we don't intend to plot.
plot_data = data[-points_to_plot:]

# Make  new figure and set the size.
fig = plt.figure(figsize=(12, 8))

# The first subplot, planning for 3 plots high, 1 plot
wide, this being the first.
ax = fig.add_subplot(311)
ax.set_title('Index Comparison')
ax.semilogy(plot_data['SP500_rebased'], linestyle='-',
label='S&P 500', linewidth=3.0)
ax.semilogy(plot_data['NDX_rebased'], linestyle='--',
label='Nasdaq', linewidth=3.0)

ax.legend()
ax.grid(False)

# Second sub plot.
ax = fig.add_subplot(312)
ax.plot(plot_data['rel_str'], label='Relative Strength,
Nasdaq to S&P 500', linestyle=':', linewidth=3.0)
ax.legend()
ax.grid(True)

# Third subplot.
ax = fig.add_subplot(313)
ax.plot(plot_data['corr'], label='Correlation between
Nasdaq and S&P 500', linestyle='-.', linewidth=3.0)
```

```
ax.legend()
ax.grid(True)
```

This is still fairly simplistic graphs of course, but it's meant to demonstrate that there is quite a bit of flexibility in how you visualize data. Now you have seen the basics, and hopefully got a feel for the way things work. We are going to make plenty of more graphs through the book.

7 Backtesting Trading Strategies

In this chapter, we will take a look at how backtesting in Python works. As we saw in chapter 4, it's extremely easy to perform simple ad-hoc tests. But that's not what we are going for in this book. We are going to do things properly.

When I started writing my first book, I decided to make sure that readers can replicate and verify everything I write. Most companies in my business

spend a lot of money on software, data and quants. I did not want to write books that only firms with large budgets could make use of. If I make claims that can't be verified, they are of little use. I wouldn't trust a book that does not show all the details and allow for replication, and it would be unreasonable for me to expect anyone else to do so.

The first thing I did was to research cheap tools that are good enough for the purpose. I did not want to tell people to go out and spend thousands of dollars a month just on market data. If you have the budget for that, fine. Knock yourself out. But it's a fair assumption that most of my readers don't.

What I ended up doing was to construct a budget quant environment and use that and only that for the purpose of my books. For my first book, *Following the Trend* (Clenow, Following the Trend, 2013), I used software that cost around 500 dollars and market data that cost a hundred or so per month. That's dirt cheap in this business. But I was surprised that quality tools can be found on the

cheap with a little research.

That being said, most backtesting platforms used by retail traders are really bad. No, I won't single anyone in particular out, but I dare to say that an overwhelming majority of what non-professionals use is horrible.

The environment I ended up using for my first book was based on **RightEdge** . It's a solid piece of software that I still use and recommend. Based on C#, it's fast and reliable, but requires quite a bit of programming knowledge to get started. C# is a great language to work in, once you get to know it. Easier than C++, it's a fast and robust language that can do just about anything you need. It's hard to proclaim any software *The Best* , and the choice often comes down to purpose and preference. It's easier to discard the bad ones until only a few good-enough remain.

I still like **RightEdge** , even though it's barely updated in a decade, and I kept using it heavily over the years and spent much time modifying it to my needs. I

relied on that same platform for the research in my second book, Stocks on the Move.

For *Following the Trend* , I used futures data from **CSI Data** , and in *Stocks on the Move* , I used data from **QuantQuote** and **Norgate Data** .

I'm sure some are wondering why I mention this, and how much money these companies are paying for it. Well, once again not getting paid for mentioning them. I simply want to be up front with what I used. Are these firms the best? I don't know, but it seems unlikely that I happened to find the very best, whatever that might mean in this context. I found these solutions to be good enough for the purpose. They also fit my requirement of low cost solutions, to allow readers to replicate the work.

I ended up needing a budget of a couple of thousand a year for the tools needed to replicate the research in those books. That seems like a good deal to me.

In Python world, you will find that practically all software is free. It won't be

as polished, or even as finished as most of us will be used to. Most Python software requires you to do a lot of work in installing, adapting and configuring before you can really use them. But free is free.

Data however tends not to be free. There are some free data sources out there, but unfortunately the trend is that they all go premium. One by one, they either cut or limit access, and offer a pay service.

You can find some basic stock data to play with, but if you want proper data that you can rely on, unfortunately free does not seem to be a robust option.

For this book, I used two data sources. For stocks, I relied on **Norgate Data** and for futures I have used **CSI Data** . Both are low cost services, with good enough quality for most daily algo modeling and they are well priced for retail traders.

Python Backtesting

Engines

The first thing that you need to understand is that there is no single way to run financial backtests in Python. Just like there is no single way to do it in C++, or other languages. A programming language is just that; a language. To use it for backtesting, we either need to construct an engine, which deals with all the details for us, or install and use an engine that someone else already made.

After all, that's what happens if you buy an off the shelf software package, such as **AmiBroker** , **RightEdge** , **TradeStation** , **NinjaTrader** , **MultiCharts** etc. The main difference in the Python world is that you are generally expected to do more work yourself. You won't get as finished, polished, visual software.

On the other hand, the flexibility and potential is much greater using Python than with most of these kind of off-the-shelf packages. Once you get into Python, you will see the benefits of being

able to do just about anything with it. And again, it's free.

Python backtesting is still in its early days. It has not yet reached mass appeal, and that can sometimes be painfully clear in terms of usability and documentation. So far, it's mostly data scientists, quant professionals and early adopters using it. But what we are seeing is that the user base is growing and, more importantly, broadening. As this continues, usability and documentation are likely to improve.

Don't expect to download an installation file, get a graphical installation program which takes care of every detail, installing a shiny Windows application where you point and click, change a few settings, run a backtest and get some colorful analysis page showing you all the details. That's not the Python way. At least not yet.

On the other hand, Python backtesting engines tend to be not only totally free, but also open source. Yes, you can get the software without paying a

penny, and you will get the full source code so that you can make any modifications that you see fit.

When you start looking for a backtesting engine for Python, you will find that there are many to choose from. It may seem overwhelming even. As you dig deeper, you will find that they are all quite different, often with their own unique strengths and weaknesses.

When I first started digging into these details, I asked friends in the financial industry who had been working with Python backtesting much longer than I had. More often than not, it was suggested that I simply build my own engine.

I can see how that makes sense for some. But I'm not looking to reinvent the wheel here. When I need to use a spreadsheet application, I simply open Excel. I don't start over, building my own spread sheet application, even though I don't necessarily agree with all the design choices that Microsoft made.

I prefer to use something that's

already built. Preferably something that I can then modify, changing things that I don't like, adding things that I see as missing. But still using the core functionality of an existing solution.

The question is then which of the many available Python backtesters to use. In this book, I'm only going to use one. I don't want to go over each and every possible Python backtester, showing code examples from all of them and leave everyone utterly confused.

Instead, I will pick a single backtesting engine and use it for all examples in this book. That way, you have a fighting chance to replicate everything from this book on your local computer, and build upon that to make your very own backtesting environment.

Zipline and Quantopian

The backtesting engine that I will use for this book is **Zipline** , developed by

Quantopian. The **Zipline** package is probably the most mature of all the currently available choices. It has a richer functionality and it scales very well to large amounts of data. The reason for this requires a brief background on the company that built it.

Quantopian is a Boston based tech and asset management firm which among other things runs a website of the same name. On that website, you'll find a totally free backtesting environment, where you can run advanced backtests on minute level data for American stocks and futures. It's all hosted, running on their servers and provides you with rich functionality and minute level data, all totally free.

At the core, Quantopian is an investment firm, albeit a little different type of investment firm. They are aiming to find untapped talent among the users who develop trading strategies on their website. If you have built something that you think is really valuable, you can submit it to their competition. If you are selected, they will allocate money to your

strategy and pay you based on performance.

It's a novel concept, and the jury is still out on this business model. Some readers may have noticed that I have appeared quite a few times as a speaker at Quantopian's conferences, QuantCon, around the world. So clearly you are now wondering just how much they are paying me to plug their company here.

I hate to disappoint but I'm not getting paid, for speaking at their conferences or for mentioning them here. Their conferences are some of the best in the field, and I show up and speak there because I enjoy it. Regarding getting paid for putting them in the book, I doubt that this publicity is worth enough to bribe an author. Don't get me wrong of course. I do work in the hedge fund field, and claiming to be morally outraged by the idea would be silly. But I sincerely question the economic value of plugging a substandard solution for a few bucks extra. Especially a free, open source solution.

If however you have a product you would want to bribe me to plug in an upcoming book, you should know that I'm partial to gold bars and Patek watches.

The Quantopian website uses the backtesting engine **Zipline** , developed and maintained by the same company. **Zipline** can also be downloaded and installed locally.

There are some important differences between using the Quantopian website and installing **Zipline** locally. Clearly, using the website is far easier. Everything is installed and set up for you. Even more importantly, you get free access to minute level data for stocks and futures. You get a nice graphical environment to work in, where you can point and click to get things done, as most people would expect these days.

Most of the trading strategies and code in this book should work just fine on the Quantopian website. But it seems like a bit of cheating, if I write a whole book around a website. In the long run, and if you are really serious about quantitative

modelling and systematic trading, you probably need a robust, local environment.

Most of the knowledge from this book can be used on the Quantopian site and for those looking to monetize their trading hobby at low risk, you may want to look closer at what they have to offer.

But going forward in this book, I will use the **Zipline** library as a teaching tool, installed locally on your own computer. Once you feel confident enough to go on your own, feel free to try other backtesting libraries and discover differences and similarities.

Having a local setup is naturally a little bit more complex. It also does not come with the aforementioned minute level free data, like the Quantopian website does. They obviously can't just give you that data to use on your local machine. Their licensing agreements with their data providers forces them to ensure that all that data stays on their server.

A result of this is that everything you do on their website needs to stay on

their website. They don't allow for saving and downloading equity curves from a trading strategy for example, among other things. If they did, then someone would quickly figure out that you can use that to siphon data out.

For a local setup, we therefore have to improvise a bit on the data side. That's actually the most complex task we have to solve. The easy solution, though hardly the best, is to use free data from a source like Quandl. Such functionality is built into the Zipline package and you can be up and running with that in minutes.

But naturally, a free internet data source is not suitable for proper backtesting. We will use Quandl data for some basic models and testing first, and later in this book we will look closer at proper data and how to hook it up.

Pros and Cons

Setting up a proper simulation environment based on Zipline without a guide can be quite painful. It may require

a bit of work to get all the pieces in place, get the right libraries installed, setting up your own data source and such. I will do my best to guide you through these parts here in this book. It could of course be argued that this issue is not necessarily a Zipline issue, but rather a larger Python issue.

The Python user base is expanding, but most code and documentation are aimed at data scientists with a strong programming background. Generally speaking, documentation in the Python community is poor. But this is likely to change as more and more people join this field.

The Zipline documentation is no exception. As far as Python documentation goes, it's not bad. But compared to what you might be used to, it's quite lackluster.

Connecting your own data source to Zipline is not entirely trivial. The documentation is near non-existing and it can be a little frustrating to get everything working. Again, this is something that I

hope to help you with in this book, and you will find full source code and samples in later chapters.

But one of the most severe downsides with Zipline is that it's only aware of a single world currency. That is, it's unaware that securities may be denominated in multiple currencies. If you only trade American stocks, that shouldn't matter. But if you trade international stocks, or global futures markets, that's a bit of a problem.

There are many great things with Zipline as well. A key point here is that, as opposed to alternative backtesting libraries, this one has a paid staff of full time developers working on it. That certainly helps drive improvements and bug fixes, and this is why Zipline has grown to such a rich set of functionality.

Compared to most current alternatives, Zipline can do more realistic simulations, handle much larger amounts of data and it's really quite fast. It also has a fairly large online community of users, and that means that it's easier to find

answers to questions when you get stuck.

Zipline works with both equities and futures, and it has some very clever features in both those areas that we will look closer at later on.

When writing a book like this, there is always a risk that something important changes after you go to print. Depending on when you read this, perhaps there is some other backtesting engine which has surpassed Zipline or perhaps something major changed with Zipline. To mitigate this risk, I will try to keep everything in this book general enough that you should be able to figure out how to apply it on a different environment as well, with as few modifications as possible.

The important point here is learning methodology. Not the exact interfaces for a specific backtesting engine.

Installing Zipline

Remember that Zipline is just another Python library. It's not a stand-alone application, where you download

some exe file and get a graphical installation program guiding you through the process. That's not how Python generally works.

If you are new to Python before this book, it might take you a little while to get used to how things are done here. Bear with me, and I will walk you through a typical Zipline installation process.

While there may be new versions of Zipline released by the time you read this, all sample code in this book assumes that you are using Zipline version 1.3. As of writing, that's the most up to date version.

First off, Zipline is designed to work either on Python 2.7 or Python 3.5. As of writing this, the most current version of Python is 3.7, and as of this date Zipline does not install on that version. Remember that in chapter 4 we discussed installing Python, and how you should install version 3.7. That's likely what you are now running by default on your computer. But fear not. There is a relatively simple way around this.

The solution is in Python's ability to

have multiple **environments** on a single computer. Think of an environment as a virtual computer, running on your actual computer. You can set up as many environments as you like, and they are completely independent of each other in every regard. They can run on different versions of Python and have different libraries installed and running.

Setting up separate environments for separate tasks is a good idea. A complex library such as Zipline, or other backtesting engines, may have very specific requirements for which versions of other libraries it needs to have installed. Perhaps it requires a specific version of Pandas, for instance.

Then imagine if you would like to try another backtesting engine. Or perhaps you want to build something completely different by yourself, where you would like the latest versions of all libraries. Now you have a conflict.

And that's why you should set up a new and separate environment for each type of activity. A backtester like Zipline

should have its own environment, so that everything can be installed and set up according to what it needs, without interfering with anything else you would like to do with Python on your computer.

Therefore, the first step here is to set up a brand new Python environment for Zipline, running on Python version 3.5. For most of this book, we are going to use this environment.

As always in Python world, there are many ways to accomplish the same thing. I won't go over every possible way, but simply pick one which I think could be easiest for beginners. If you know of other ways, feel free to use them. But I will primarily use the **Anaconda** way, using as much graphical user interface methods as I can.

Go back to **Anaconda Navigator** . We are going to use it to create a new Python environment. Click on the Environments menu on the left. Depending on your exact installation, you will likely just see one environment called root or base. At the bottom, you see some

buttons to create, clone, import and remove environments. Click create.

We are going to call this new environment zip3 5 , so that we remember what it's for and which version it's running. Be sure to select Python version 3.5 in the dropdown, as in Figure 7-1.

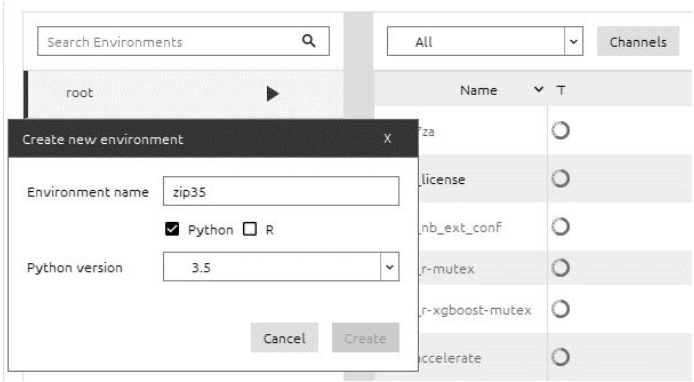


Figure 7-1 Creating Zipline Environment

Now you should see your new zip3 5 environment listed right there, under root. We are now ready to install the actual Zipline library.

In the previous chapter we saw how you can install libraries in **Anaconda Navigator** . You could install Zipline the same way, as long as you add the Quantopian channel. In Figure 7-1 you can see the button for adding a channel.

Think of a channel as a place where Python libraries can be downloaded from. In this case, Quantopian made their own channel.

Remember that you have to select the environment that you want to change before adding or removing any libraries. In this case, we want to add Zipline to the zip3 5 environment, so you need to select that first to see the libraries available.

While you can install Zipline by just adding the Quantopian channel, finding Zipline and checking the Zipline library, you could also use the terminal if that's what you prefer. Most readers are probably happy to do things visually, but I want to mention this possibility so that you at least have heard of it.

The other way of installing a library would first involve opening the terminal window. The terminal is a text input, command line environment and thereby not the overly user friendly, but it can be useful to understand. You can open it by clicking that triangle next to the environment names, as you can see in

Figure 7-1.

If you click the button next to your newly created environment and select Open Terminal, you should get a terminal window, with the zip3 5 environment already activated for you. That means if we now give a command to install a Python library, it will install in that environment for us.

To install Zipline in the terminal, type the following command.

```
conda install -c Quantopian zipline
```

This tells the **conda** installation program to check the Quantopian channel for a library called Zipline, and install it.

The Zipline library comes with a list of dependences. That is, other libraries of specific versions that it relies upon to function. The **conda** installation program is aware of this, and will ask you if it's fine for you to go ahead and install all of those.

Once this process is finished, you now have Zipline installed on your computer. Of course, that's just one of

several steps to actually get to the point of trying out a backtest.

Problems with Installing Zipline

The instructions above for how to install the Zipline library have been tested by myself and a few others on different installations, operating systems and such. But with a complex library like this, there is always the potential for curveballs. If for some reason your Zipline installation did not work as planned and you got some sort of odd error message when attempting it, the best place to look for tips and help would be either Zipline's GitHub page, **https://github.com/quantopian/zipline/issues** , or the Google Groups forum, **https://groups.google.com/forum/#!forum/zipline** .

Possible Version

Issue

After the first release of this book, a new version of Conda was released which broke some dependencies and refused to install Zipline. If you have Conda version 4.7 or higher and had trouble installing Zipline, you may want to try the workaround below.

```
conda config – allow_conda_downgrades true
conda install conda = 4.6.11
```

Zipline Installation Issue

A reader of the first edition of this book reported the following issue and solution.

Error:

```
Collecting package metadata
(current_repodata.json): done
Solving environment: failed with
current_repodata.json, will retry with next
repodata source.
```

Solution:

```
conda update – n base – c defaults conda
conda config – set restore_free_channel true
conda install – c Quantopian zipline
```

Patching the Framework

Zipline, like most software in Python world is not only free, but open source. You can see and edit all the code as you like, and one big advantage of this is that we can make changes and even fix bugs when needed. You may wonder why you would ever want to go and edit someone else's code, but it can be very useful to have this possibility.

While writing this book, an interesting example of such a situation came up. One day, all the backtests suddenly stopped working, with an error message about failing to load benchmark data from some web address. As it turns out, there is a web call executed during backtests, where Zipline tries to pull data

from an online source. This source, much like what happened to many other free data sources, suddenly stopped working without warning.

As of writing, this issue has not yet been addressed. But no worries, we can do it ourselves easily. We don't actually need this benchmark data for anything that we will do in this book, and the easiest fix is to simply remove the web call and just return empty data.

My version of Zipline is 1.3, and if you're working with the same version, you likely need to do this fix manually. It won't take long.

First locate a file called **benchmarks.py** , which should be in your Zipline installation folder. You can use the file system search tools to locate it. If you are on Windows, the path will probably be similar to this.

C:\ProgramData\Anaconda3\envs\zip35\Lib\site-packages\zipline\data

It's always a good idea to keep a backup of the original code, so rename this file to keep a copy. Then make a new

file, with the name **benchmarks.py** , and put the following code in it before saving.

```
import pandas as pd
from datetime import datetime
from trading_calendars import get_calendar

def get_benchmark_returns(symbol):
    cal = get_calendar('NYSE')
    first_date = datetime(1930,1,1)
    last_date = datetime(2030,1,1)
    dates = cal.sessions_in_range(first_date, last_date)
    data = pd.DataFrame(0.0, index=dates,
columns=['close'])
    data = data['close']
    return data.sort_index().iloc[1:]
```

This code will simply return a data series from 1930 to 2030 with all zero values, which will effectively sidestep this web call issue, with no negative impact.

You may also need to bypass the cache in **loader.py** , in the same folder:

```
    """
    if data is not None:
        return data
    """
```

Zipline and Data

As mentioned earlier, Zipline is the same backtesting engine used on the Quantopian website, but there are some pretty important differences. The most important being how you actually get financial data.

On the Quantopian site, minute level data for American stocks and futures is included for free. It's just there, ready to use. Not so with a local Zipline installation. It's fairly easy to get basic, free stock data hooked up. It's trickier to connect Zipline with high quality data from your source of choice.

In this book we are going to do both. To begin with, we will use freely available data off the internet. I will explain why this may be easy and convenient, but probably not a good idea for proper backtesting. Later in the book, we will look at how to get your own, custom data working.

When it comes to Zipline and data, there are two words that need to be explained, which are core to this library's terminology; bundle and ingest. These are

Zipline specific terms, and if you use other backtesting engines they are most likely not applicable.

A bundle is an interface to import data into Zipline. Zipline stores data in its own preferred format, and it has good reason for doing so. Zipline is able to read data incrementally, and only hold a part of it in memory at any given time.

If you run a backtest on a few stocks on daily data, that does not matter one bit. But if you are running it on a few hundred stocks on minute level, you are dealing with vast amounts of data. Reading it all and keeping in memory is not an option.

Zipline's solution is to import all data first, and store in this special format which allows for incremental reading. The bundle is the interface, which reads data from an actual data source and hands it over to Zipline for processing and storage.

Every data source needs its own bundle. There are some basic bundles included, but once you start getting really serious about backtesting, you probably

want to write your own. We will get to that.

Then there is the second word, ingest. That word refers to the process of reading data with the help of a bundle, and storing in Zipline's own format. One needs to ingest a bundle before one can run a backtest. You do this from the terminal, running a command which tells Zipline to use a specific bundle to read data and store it so that it will be ready for use by a backtest.

The way that Zipline handles data is at the same time a core strength and a core weakness of this backtesting library. It's an advanced method, allowing for fast access to vast amount of data. It scales exceptionally well and you can run highly complex strategies on extremely large data sets. That's something which can't be said for any alternative library that I have tried.

But on the other hand, this advanced functionality comes at a price. To make proper simulations, you need proper data. This is likely to entail a

purchase of commercial data, and you will be left to your own devices to construct a bundle which can import to Zipline.

For now, it's only important that you understand what these two words mean. Later on, we will take a closer look at bundles and how to construct them.

Ingesting the Quandl Bundle

How is that for a headline? In case you are standing in the book store, considering buying a trading book and flipped to this particular page, you would be understandably confused.

Quandl is a financial data provider, or perhaps aggregator is a better term, where you can download data off the internet. Some of their data is free, while much of it requires a subscription.

It used to be easier to get free, basic stock market data from the internet. If you pick up a book on the topic of financial modeling that was written before

mid-2017, you likely see examples of how easy it is to get this data from Yahoo Finance, or perhaps Google Finance. Yes, this was indeed very easy and useful for all kinds of purposes. It was never considered high quality professional financial data, but that's beside the point. It was good enough for ad-hoc use.

But by mid-2017, the two companies which had been providing free of charge API access to financial data, suddenly and without warning stopped doing so. If they ever got around to explaining why, that explanation has still eluded me.

This little history of free online stock data is good to know, as you are likely to run into it again. Either if you read an older book, or if you go search the internet for examples. Everyone was using these sources, up until they suddenly stopped working.

One source which still is free, at least for now, is the basic Quandl access. Zipline has an included (bundled?) bundle, which reads equity data from

Quandl, and that's what we will use for our initial examples.

To be able to use the free Quandl data, you need to have register an account with them. No, it does not cost anything. Head over to Quandl.com and set up a free account. Once that's done, you can find your API key in your account settings. Copy that key out. We need it to be able to download data.

Next, go to the terminal for the zip35 environment. Remember that you can launch it from Anaconda, in the environments screen. Run the following command.

```
Set QUANDL_API_KEY = your_own_api_key
```

Make sure to put your own key from the settings on **Quandl.com** . Setting this key only needs to be done once on your computer. After this, the setting is there and will be remembered. After this we are able to download data, and to do that we need to ingest the Quandl bundle.

```
zipline ingest – b quandl
```

This runs the ingest process, using

the bundle called quandl to download free stocks data off the internet and store locally. You should see the progress bar moving, and it may take a few minutes to complete.

After this, you have access to daily equity prices to use for your backtests. Of course, this is a once off process, and it will not automatically update tomorrow. You will have to repeat this process when you want fresh data. Or figure out a way to automate it.

In case the Zipline command did not work and you were instead left with the less than helpful error message ‘ Failed to create proces s ’, there is an easy fix to this known bug. This is a bug that occurs on Windows environments only, and only if there are spaces in the path to your python.exe file. As of writing this it was not yet fixed. You need to locate a file called **zipline-script.py** , which is likely located either in your user profile or in the program data folder. The exact location depends on your local installation, but if you did get this particular error, search for the file and

you will find it.

Open that in Notepad or similar, and put quotes around the path to python.exe.

If the top line of that file looked like this:

#!c:/path to/your/python interpreter/python.exe

Change it to this:

#!"c:/path to/your/python interpreter/python.exe"

Trouble Ingesting Quandl

Feedback from the first edition shows that a small number of readers failed to ingest quandl with the following error:

"ValueError: Boolean array expected for the condition, not float64"

The same readers report that the issue was solved by downgrading Pandas to 0.18.1.

Installing Useful Libraries

In chapter 4 we installed some very useful libraries, which can help get things done easier in Python World. When you install libraries, they will only install in one specific Python environment. Back then, we only had one environment; the default root environment. Now, in this chapter, we have created a new environment specifically for Zipline. That means that the libraries we installed before, are not installed on our zip3 5 environment. We need to do that again.

You could do this exactly the same way as we did it in the previous chapter, and if you prefer that, go right ahead. But I will show you another, more visual way to install libraries for those who prefer that.

Go to **Anaconda Navigator** , click on Environments and then on the zip3 5

environment that we created. On the right side of the screen, you now see the installed libraries for this environment. But if you change that dropdown on the top to read *Not Installed* , you will see available but not yet installed libraries.

Do that and scroll down to **matplotlib** . Check the box, as in Figure 7-2. Now do the same with **nb_conda** . These were the libraries we installed in chapter 4. Once you have selected them, just hit the Apply button at the bottom, and they will be installed in your zip3 5 environment.

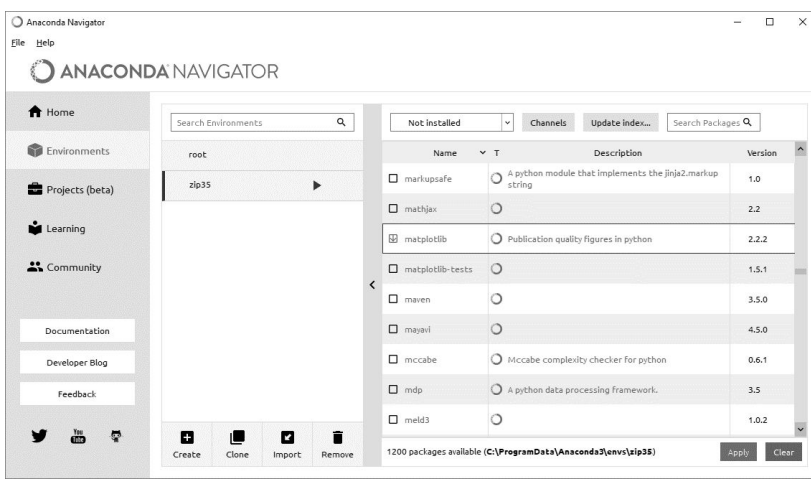


Figure 7-2 GUI Library Installation

Where to Write

Backtest Algos

As is often the case with Python, there's more than one way to do the same thing. While it's possible to use a variety of software solutions to write your Python backtests, I will stick to Jupyter Notebook, as we used in the previous chapters of this book.

Jupyter is very well suited for this type of tinkering, that backtest development usually entails. It's a great environment for testing code, and quickly getting the output and results in front of you.

One thing to keep in mind when working with Jupyter is to make sure that you have the correct environment active. After we installed the **nb_conda** library in the previous section, this is easily done.

To launch Jupyter Notebook in the new zip3 5 environment, start by opening **Anaconda Navigator** , and go to the Environments view. Click the triangle next to the zip3 5 environment, as you can see in Figure 7-2 and select Open with

Jupyter Notebook.

Once you are in the Jupyter interface, you can make a new notebook with the zip35 environment by selecting it in the New dropdown, as in Figure 7-3.

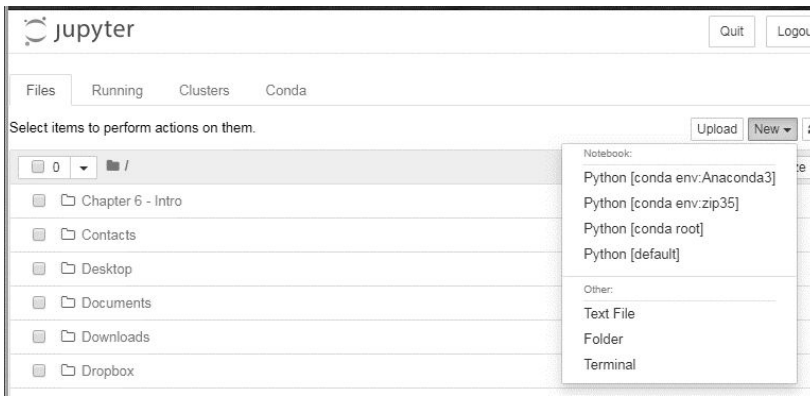


Figure 7-3 Making a new notebook

Your First Zipline Backtest

For this first trading algorithm, we are not going to worry about realism or viability of the strategy. For now, we are merely looking at the basics of creating an algorithmic backtest.

The strategy we will create first will trade a single stock; Apple. We are going

to be long Apple if the price closes over its 100 day moving average. If not, we will be in cash. The data will come from Quandl in this case, so we will be using the Quandl bundle, as discussed in the previous chapter.

The actual logic for this strategy is so simple that the bulk of the code we are going to write for it will be about showing the results, not computing the strategy. For showing the results, we will use a very similar approach to what was described in the previous chapter for how to make prettier graphs.

As with earlier, I will show code segments bit by bit as I describe them, and then show the complete code in one go after the explanations.

To begin with, as always, we need to import various libraries that we intend to use. Python alone, without importing libraries, is quite limited and to perform complex operations such as backtesting, we need a few additional things.

For a simple model like this, we don't need many import statements. Every

time we run a Zipline backtest, we will need to import run_algorith m . We will also need a few things from the zipline.ap i , but which exact things we need from there may vary depending on what we want to do in the backtest. In this case, we need the ability to place trading orders in terms of target percent, as well as the ability to look up stock symbols based on the ticker. The methods order_target_percen t and symbo l can do that for us. Later on, you will see many more methods that we can get from this library.

```
# Import Zipline functions that we need
from zipline import run_algorithm
from zipline.api import order_target_percent, symbol
```

We also import the datetim e functionality as well as pyt z which lets us define time zones. Both of which are needed when we tell Zipline when exactly to start and end the backtest.

```
# Import date and time zone libraries
from datetime import datetime
import pytz
```

The final import statement is for matplotli b , so that we can draw some graphs based on the results of the

backtest.

```
# Import visualization
import matplotlib.pyplot as plt
```

There are three functions in the code for this algorithm, and you will see the same set of three functions many more times in this book. These are the functions initialize , handle_data and analyze .

The function initialize will be executed before the backtest starts and this is where we will set parameters and prepare what needs to be prepared before the run. In this section, as you see in the code segment just below, we do two things. We set the stock that we want trade and we set the moving average window that we want to use. Those are the only two settings that we need for this backtest.

You can see in the code below how we look up the stock symbol based on the ticker, using the function symbol , which we imported just now.

```
def initialize(context):
    # Which stock to trade
    context.stock = symbol('AAPL')
```

```
    # Moving average window
    context.index_average_window = 100
```

The next function is handle_dat a , which is executed for each new data point. As we are dealing with daily data here, this function will be called once per day. This is where the trading logic goes. The rules for when to buy, when to sell, and how much. It's all done right here, in handle_dat a .

The first thing that we do here in handle_dat a is to pull time series history. This is an important concept, as you will almost always need to do this for your backtests.

```
def handle_data(context, data):
    # Request history for the stock
    equities_hist = data.history(context.stock, "close",
                                 context.index_average_window,
"1d")
```

As you can see in the code above, we can use a function called histor y , which is part of the dat a object that was passed to us in the function definition. This function can pull history for a single symbol or for many at the same time. In this case, we are supplying just a single

stock, but we could have provided a whole list of stocks as the first argument if that's what we were interested in.

As the second argument to the function history, we here supply the string "close" . That's because all we need for this model is a single field, the last closing price. Again here, we could have supplied a list of strings, such as ['open','high','low','close'] if we wanted to.

Next in the same history function, we specify how many data points we want. Well, all we need to do with the data in this model is to calculate a moving average. And we already know how many periods that moving average window is, don't we? Remember that we stored that value in the variable context.index_average_windo w .

Lastly, we supply the string "1d" to tell the history function that we are looking for daily frequency, one day interval.

Now that we have the historical data available, it's time for the trading logic. Do you still remember the rules? We

want to be long if the price is above the moving average, else flat. That's it.

```
# Check if price is above moving average
if equities_hist[-1] > equities_hist.mean():
    stock_weight = 1.0
else:
    stock_weight = 0.0
```

This code segment uses the `if` statement, checking whether the price is above the average price for the requested time series history. As we requested the same amount of data points as the moving average period, all we need to do is to calculate the mean of those prices. No need to actually "move" the average, is there. Also remember the importance of indentation, and how we need that initial tab after the `if` and after the `else`.

In that segment, we don't actually trade. Not yet. We just set the value of the variable `target_weight`.

The trading is next, in the final row of `handle_data`.

```
order_target_percent(context.stock, stock_weight)
```

We are using an order method here called `order_target_percent`. This is a handy

method if you want to automatically target a certain percent exposure. You could also calculate number of shares desired, and use the method order_targe t if you so prefer, which would then need to be imported at the top as well, just like order_target_percen t is now.

Finally, there is the function analyz e . This function will be called after the backtest is all done, and this is where we will calculate analytics and visualize results. When this function runs, we are passed the objects contex t and per f , which will contain all the information we need about the backtest results. In the next chapter we will look in more detail on what can be done with these.

At the bottom of the code sample, we set the start and end date, before starting the backtest off. Note the input parameters used when starting the backtest run.

```
# Set start and end date
start_date = datetime(1996, 1, 1, tzinfo=pytz.UTC)
end_date = datetime(2018, 12, 31, tzinfo=pytz.UTC)

# Fire off the backtest
```

```
results = run_algorithm(
    start=start_date,
    end=end_date,
    initialize=initialize,
    analyze=analyze,
    handle_data=handle_data,
    capital_base=10000,
    data_frequency = 'daily', bundle='quandl'
)
```

This is where we tell the Zipline engine when to start and when to end the backtest. We tell it which functions to run before, during and after the backtest and how much capital to start off with. Finally we also define the data frequency as well as which data bundle to use.

Here is the complete source code for this first Zipline backtest. As with all source code in this book, you can also download it from the book website.

```
# This ensures that our graphs will be shown properly
in the notebook.
%matplotlib inline

# Import Zipline functions that we need
from zipline import run_algorithm
from zipline.api import order_target_percent, symbol

# Import date and time zone libraries
from datetime import datetime
```

```
import pytz

# Import visualization
import matplotlib.pyplot as plt

def initialize(context):
    # Which stock to trade
    context.stock = symbol('AAPL')

    # Moving average window
    context.index_average_window = 100

def handle_data(context, data):
    # Request history for the stock
    equities_hist = data.history(context.stock, "close",
                                 context.index_average_window,
"1d")

    # Check if price is above moving average
    if equities_hist[-1] > equities_hist.mean():
        stock_weight = 1.0
    else:
        stock_weight = 0.0

    # Place order
    order_target_percent(context.stock, stock_weight)

def analyze(context, perf):
    fig = plt.figure(figsize=(12, 8))

    # First chart
    ax = fig.add_subplot(311)
```

```
    ax.set_title('Strategy Results')
    ax.semilogy(perf['portfolio_value'], linestyle='-',
                label='Equity Curve', linewidth=3.0)
    ax.legend()
    ax.grid(False)
```

```
    # Second chart
    ax = fig.add_subplot(312)
    ax.plot(perf['gross_leverage'],
            label='Exposure', linestyle='-', linewidth=1.0)
    ax.legend()
    ax.grid(True)

    # Third chart
    ax = fig.add_subplot(313)
    ax.plot(perf['returns'], label='Returns',
linestyle='-.', linewidth=1.0)
    ax.legend()
    ax.grid(True)

# Set start and end date
start_date = datetime(1996, 1, 1, tzinfo=pytz.UTC)
end_date = datetime(2018, 12, 31, tzinfo=pytz.UTC)

# Fire off the backtest
results = run_algorithm(
    start=start_date,
    end=end_date,
    initialize=initialize,
    analyze=analyze,
    handle_data=handle_data,
    capital_base=10000,
    data_frequency = 'daily', bundle='quandl'
)
```

Once you execute this code, it might take a minute or so, and then you should get the results popping up below, in the shape of three charts. In the code segment, you should be able to see how we create the three charts one by one. The first one shows the equity curve, the value development of our fictive portfolio.

The second chart shows the exposure, which the Zipline API prefers to call leverage. Lastly, the third chart shows the daily percentage returns.

You should now see a figure such as

Figure 7-4. Some basic outputs, showing us roughly what this simple algorithm would have done for us, in theory of course.

In case you are underwhelmed at this point by the details of the output, fear not. A key strength of Python is in analyzing time series, and once the backtest is done, that's what this is all about. Analyzing the output time series. From here, we have plenty of options on how to slice, dice and display the data.

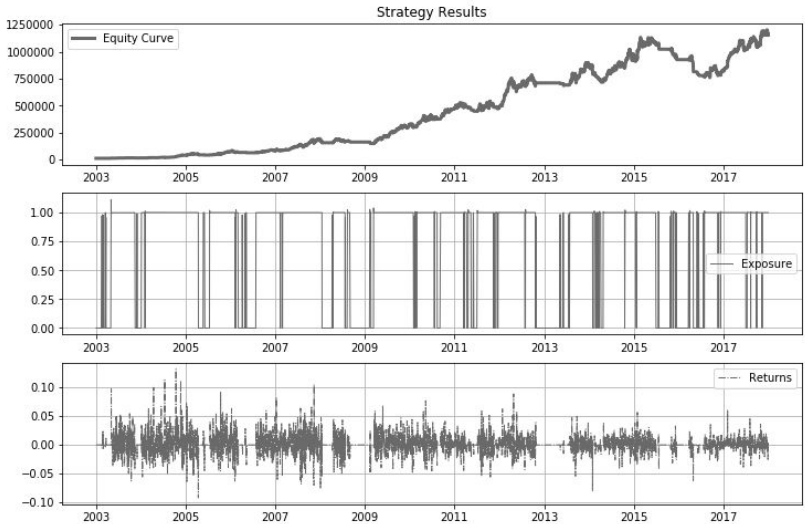


Figure 7-4 First Backtest Output

Portfolio Backtest

In the previous backtest, we used a very simple strategy on a single instrument. That's probably not how most of you tend to operate. There used to be a clear divide between the hobby segment and industry participants in that the former tended to focus on single model, single asset strategies. That's not necessarily the case anymore, as increasingly more advanced tools have become available to the general public.

Not long ago, the only tools

available to the public were either exclusively single asset strategy platforms, or had some rather shoddy after market portfolio extension slapped on top of it as an afterthought.

The core problem with single market strategies is that there is no diversification possible. They become pure market timing models, and such things rarely work out in real life. Diversification is absolutely key to professional trading.

Another issue with single market strategies is the selection of that single market. Your trading rules may be systematic, but your selection of a single stock or other asset to trade is clearly discretionary. Picking that one asset is the single largest design choice of your entire strategy.

After all, the only reason why I picked Apple for the previous example is that I know that it would show great results. A long only trend following logic on a stock which has had a tremendous bull run all but ensures a great looking backtest. But that does not necessarily

have any predictive value.

When constructing equity models, it's of vital importance to properly deal with instrument universe selection. That is, you need to consider carefully which stocks your algorithm will include. To have a fighting chance of showing some predictive value, your instrument selection needs to be realistic. If you test a strategy ten years back in time, the instrument selection ten years ago in your backtest need to resemble what you would have reasonably selected in reality back then.

The worst possible way to select stocks for a backtest is to pick those that are hot right now. They are of course hot now because they had great performance, and that means that your backtest is severely distorted before it even started.

In chapter 11 we will start using realistic investment universes, which aim to minimize the risk of bias. But before we get there, we are going to construct a portfolio backtest, just to get a feel for the mechanics of working with multiple

markets.

In fact, what we are about to do here will have a serious logical flaw, quite on purpose. But let's just get on with it, and see if you can spot the logical error before I explain it.

Our first portfolio simulation will trade the index constituents of the Dow Jones Industrial Average. For each of the 30 member stocks, we'll check each day if the price is above or below its respective 100 day moving average.

If the price is above the moving average, we will be long with a notional allocation of 1/30, or about 3.33%. That means that in a perfect bull market, if all 30 stocks are above their respective moving averages, we will have an overall portfolio exposure of 100 percent, but most of the time it will be below that.

In this example, we are going to define the stocks that we want to consider for trading, our investment universe, in the initializ e function. As you see here, we again use the contex t object and attach a list of items to it, to be read later. Some

readers who are familiar with programming may ask if we couldn't just use global variables for this, and that would be fine as well if you so prefer.

As we have seen in the previous example, Zipline makes a difference between the symbol object and the ticker string. What we need is a list of the symbols, but to make the code easier to read and edit, I first listed the symbol strings, and then in a single row I made a new list of all the corresponding symbol objects.

```
def initialize(context):
    # Which stock to trade
    dji = [
        "AAPL",
        "AXP",
        "BA",
        "CAT",
        "CSCO",
        "CVX",
        "DIS",
        "DWDP",
        "GS",
        "HD",
        "IBM",
        "INTC",
        "JNJ",
        "JPM",
```

```
    "KO",
    "MCD",
    "MMM",
    "MRK",
    "MSFT",
    "NKE",
    "PFE",
    "PG",
    "TRV",
    "UNH",
    "UTX",
    "V",
    "VZ",
    "WBA",
    "WMT",
    "XOM",
]
```

```
# Make a list of symbols from the list of tickers
context.dji_symbols = [symbol(s) for s in dji]
```

In the handle_dat a function, we're going to make use of some Pandas tricks to get the logic done easily. First we need to fetch the historical data, and we have seen this before. This time however, we are getting history for all the stocks at once.

```
# Get history for all the stocks
stock_hist = data.history(context.dji_symbols,
"close", context.index_average_window, "1d")
```

Next we'll do some **Pandas** voodoo. What I plan to do here is to make a **DataFrame** with a column to tell us if a stock is above or below the average, and a column for what percentage weight of the stock that we want to hold. I'm doing it this way to teach you how this can be done with Pandas.

First we create a new **DataFrame** and then we make a column called above_mea n , which will be set to **True** if the stock is above, and otherwise it will be **False** . Note how this logic is done in a single, clean row below. We are comparing the last price with the mean price, and we can find the last price with the ilo c function, which can locate lows in a **DataFrame** based on their numerical position. Minus one in this case means the last one, i.e. the latest point.

```
# Make an empty DataFrame to start with
stock_analytics = pd.DataFrame()
```

```
# Add column for above or below average
stock_analytics['above_mean'] = stock_hist.iloc[-1]
> stock_hist.mean()
```

We continue to use the same

DataFrame that we just created, and add a column called weigh t . Now here is an interesting trick to pay attention to. This time we are using the function lo c which can locate rows in a **DataFrame** based on a logical criterion.

That first row below does the following. It locates rows where the column above_mea n is **True** , and for those rows, it sets the column weigh t . As we said before, we want the weight to be 1 divided by total index stocks. This way of locating rows and setting a value can get things done quickly and easily. Consider the alternative of looping through all the rows one by one.

We do the exact same thing for rows where the price is below the average, and set the weight to zero.

```
  # Set weight for stocks to buy
  stock_analytics.loc[stock_analytics['above_mean']
= = True, 'weight'] = 1/len(context.dji_symbols)
```

```
  # Set weight to zero for the rest
  stock_analytics.loc[stock_analytics['above_mean']
= = False, 'weight'] = 0.0
```

Now we know which stocks to be

long and which to be flat, and we know the weights. Now we can loop and make the trades, one by one. We can use the .iterrrows( ) to get the index and corresponding **DataFrame** row, one at a time.

As a safety, I added a check for if the stock can be traded at this time.

```
# Iterate each row and place trades
for stock, analytics in stock_analytics.iterrows():
    # Check if the stock can be traded
    if data.can_trade(stock):
        # Place the trade
        order_target_percent(stock, analytics['weight'])
```

This is really not that many actual lines of code. As you see, the trading logic could be done in just a few simple statements. This is exactly what you want with a backtesting environment. You want to focus on what's important. You want to spend your time testing your trading ideas, not writing page after page of code just to get simple stuff done. Python, with some help from Zipline and Pandas can achieve this for you.

```
# This ensures that our graphs will be shown properly in the notebook.
%matplotlib inline
```

```
# Import a few libraries we need
from zipline import run_algorithm

from zipline.api import order_target_percent, record,
symbol
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import pandas as pd

def initialize(context):
    # Which stock to trade
    dji = [
        "AAPL",
        "AXP",
        "BA",
        "CAT",
        "CSCO",
        "CVX",
        "DIS",
        "DWDP",
        "GS",
        "HD",
        "IBM",
        "INTC",
        "JNJ",
        "JPM",
        "KO",
        "MCD",
        "MMM",
        "MRK",
        "MSFT",
        "NKE",
```

```
        "PFE",
        "PG",
        "TRV",
        "UNH",
        "UTX",
        "V",
        "VZ",
        "WBA",
        "WMT",
        "XOM",
    ]
```

```
    # Make a list of symbols from the list of tickers
    context.dji_symbols = [symbol(s) for s in dji]
```

```
    # Moving average window
    context.index_average_window = 100
```

```
def handle_data(context, data):
    # Get history for all the stocks
    stock_hist = data.history(context.dji_symbols,
"close", context.index_average_window, "1d")
```

```
    # Make an empty DataFrame to start with
    stock_analytics = pd.DataFrame()
```

```
    # Add column for above or below average
    stock_analytics['above_mean'] = stock_hist.iloc[-1]
> stock_hist.mean()
```

```
    # Set weight for stocks to buy
```

```
    stock_analytics.loc[stock_analytics['above_mean']
== True, 'weight'] = 1/len(context.dji_symbols)
```

```
    # Set weight to zero for the rest
    stock_analytics.loc[stock_analytics['above_mean']
== False, 'weight'] = 0.0
```

```
    # Iterate each row and place trades
    for stock, analytics in stock_analytics.iterrows():
        # Check if the stock can be traded
        if data.can_trade(stock):
            # Place the trade
            order_target_percent(stock, analytics['weight'])

def analyze(context, perf):
    fig = plt.figure(figsize=(12, 8))
```

```
    # First chart
    ax = fig.add_subplot(311)
    ax.set_title('Strategy Results')
    ax.plot(perf['portfolio_value'], linestyle='-',
            label='Equity Curve', linewidth=3.0)
    ax.legend()
    ax.grid(False)
```

```
    # Second chart
    ax = fig.add_subplot(312)
    ax.plot(perf['gross_leverage'],
        label='Exposure', linestyle='-', linewidth=1.0)
    ax.legend()
    ax.grid(True)
```

```
    # Third chart
    ax = fig.add_subplot(313)
    ax.plot(perf['returns'], label='Returns',
linestyle='-.', linewidth=1.0)
    ax.legend()
    ax.grid(True)

# Set start and end date
start = datetime(2003, 1, 1, tzinfo=pytz.UTC)
end = datetime(2017, 12, 31, tzinfo=pytz.UTC)

# Fire off the backtest
results = run_algorithm(start=start, end=end,
                        initialize=initialize,
analyze=analyze,
                        handle_data=handle_data,
                        capital_base=10000,
                        data_frequency = 'daily',
bundle='quandl' )
```

After running this code, you should have an output showing you something similar as Figure 7-5.

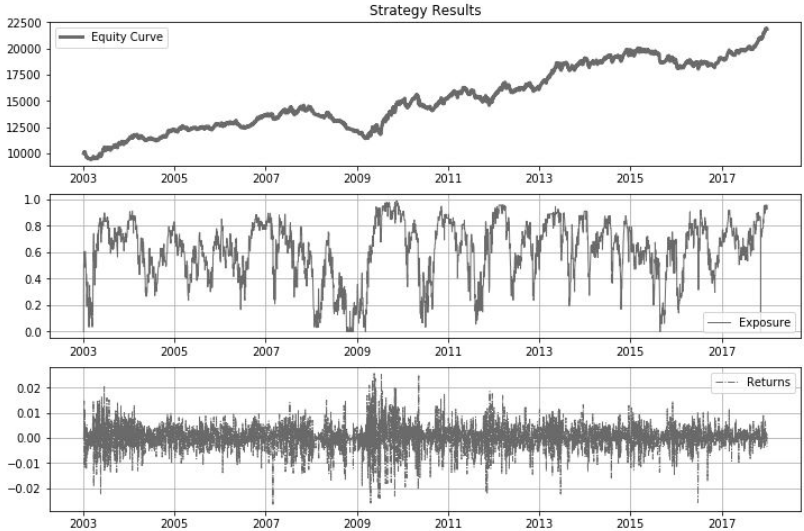


Figure 7-5 First Portfolio Backtest

Now I told you before that there is some sort of an issue with this backtest, and I suspect most readers have already picked up on that. If you did not spot it yet, go back and look at the code a moment.

I'm not referring to the fact that we have no trading costs, no slippage, or even to the viability of the strategy as such. It's not even that we started the backtest in 2003 just to make it look better, or that we magically trade instantly on the same close price that we use for the calculation. No, there is a much more fundamental issue.

What we did here, albeit on purpose, was to commit a very common but equally serious offence. We specifically set the investment universe to the current constituents of the Dow Jones Index. The index did not have the same members back in 2003 when this simulation starts.

Why does that matter? Consider why these stocks are in the index in the first place. Like for any index, the stocks that are in the index are only there because they performed well in the past. Of course a strategy which relies on buying stocks will perform better if you run it on stocks which we already know had large gains in the past.

This is a very important issue, and we will return to it again in this book.

Data Used for this Book

To be able to build proper backtests and experiment with your trading ideas,

you first need to obtain financial data of sufficient quality and import it so that your backtesting software can read it. This can be a non-trivial task, but it's something that has to be taken seriously. In the interest of maintaining the flow of this book, and to trick you into reading the more interesting parts first, I will explain data handling in more detail in chapters 23 and 24.

There you will be able to read in more detail about how to make Zipline function with your own data, and you will find source code and explanations. I suspect that most readers will want to read through the book first, before constructing the full scale backtesting environment and trying out all the source code. But I will show you all the code that you need.

For this book, I used equity data from **Norgate Data** and futures data from **CSI Data** . I picked those two because I know that they are good, and they are both priced in a range accessible to hobby traders and beginners.

8 Analyzing Backtest Results

Running a backtest and analyzing the results are two quite different tasks. In chapter 7 we wrote some simple backtests, but we did not bother much with analyzing the results.

Analyzing time series is a core strength of Python and we have no shortage of options of what to do with the backtest results. It would be pointless to ask if we can calculate your favorite analytic, because we can of course calculate and visualize just about anything we can think of.

There are two ways we could approach analyzing the backtest results. We could install and use purpose built libraries, or we could build something more custom. In this chapter, we will look at both possibilities and aim to provide you with the tools and knowledge that you will need to figure out if your

backtest is any good or not. Simply plotting an equity curve is easy enough, but it rarely tells enough of a story to serve as the basis of real decision making.

Installing PyFolio

The makers of the Zipline backtesting library have also made a useful library to analyze backtest results. This library, **PyFolio** , makes for a good entry gateway to backtest analysis. For many, it may be the only thing you need. For others, it may be a good start and something you can learn from and improve upon.

To install **PyFolio** , we need to use a different method than we have been using before. In the interest of keeping things consistent, I would have preferred to keep using the same way of installing libraries, but as of writing this, a technical issue prevents it.

Instead, we will use a different installation method. Again, open the terminal with the `zip3 5` environment

activated. The easiest way is through **Anaconda Navigator** , selecting the correct environment and then starting a terminal window from there. Then enter the following to install **PyFolio** .

```
pip install pyfolio
```

As you explore later on your own, you will find that **PyFolio** can do quite a bit of reporting for you. As a demonstration, we are going to create something which **PyFolio** calls a returns tear sheet. That is, an overview analysis of the returns of the strategy. To do this, we first need an algorithm, a trading strategy to analyze.

Portfolio Algorithm to Analyze

We could of course take one of the algorithms that we looked in in the previous chapter, but that wouldn't be much fun. Instead, I will take the portfolio model on the same static Dow Jones that we constructed earlier, and change things

around a little.

By using a different trading logic, I can take this chance to teach you something new on that side as well. There are some neat little tricks that I'd like to show you along the way, as we construct a new variant of our Dow Jones trading model.

The model that we will use for analyzing results will also be based on the Dow Jones stocks, but it will use a simple momentum approach. We will check the trading rules only once per month this time, to create a long term investment algorithm which does not have excess costs in terms of commission and tax.

The Dow Jones Industrial Average consists of 30 stocks. Our algorithm here will measure the return in the past month for these 30 stocks, sort them from best to worst performance and buy the top 10. They will be equal weighted again, so each stock would have a weight of 10%.

Fire up a new **Jupyter Notebook** for your zip3 5 environment, and build away. If you already feel comfortable

enough with Python, go ahead and try to create these rules yourself. It's not that hard.

On top of writing these new trading rules, there is also the small matter of the actual purpose of this chapter; analyzing backtest results. For that, we need to import the **PyFolio** library that we just installed.

By now you may have noticed the seemingly odd object called contex t . This is a Zipline specific convenience. You can add anything you like to this context object, and it will stay persistent so that you can access it later on. In this case, I tag on a list of stocks to trade and some model settings in the initialize routine. I then read these settings again in the daily trading logic. Note how the context object is passed to our scheduled handle_dat a routine.

Those familiar with programming may ask why we are not just putting these settings up top, as global objects. Sure, that works fine too, if that's what you prefer. In fact, I do prefer that myself as it

tends to result in code that's easier to read and modify, and I will show you how that works in later code in this book. For now, I just wanted to show how you can use the context object, in case you find it of use.

```
# Import a few libraries we need
from zipline import run_algorithm
from zipline.api import order_target_percent, symbol,
\
    schedule_function, date_rules, time_rules
from datetime import datetime
import pytz
import pyfolio as pf

def initialize(context):
    # Which stocks to trade
    dji = [
        "AAPL",
        "AXP",
        "BA",
        "CAT",
        "CSCO",
        "CVX",
        "DIS",
        "DWDP",
        "GS",
        "HD",
        "IBM",
        "INTC",
        "JNJ",
        "JPM",
        "KO",
        "MCD",
```

```
        "MMM",
        "MRK",
        "MSFT",
        "NKE",
        "PFE",
        "PG",
        "TRV",
        "UNH",
        "UTX",
        "V",
        "VZ",
        "WBA",
        "WMT",
        "XOM",
    ]
```

```
    # Make symbol list from tickers
    context.universe = [symbol(s) for s in dji]
```

```
    # History window
    context.history_window = 20
```

```
    # Size of our portfolio
    context.stocks_to_hold = 10
```

```
    # Schedule the daily trading routine for once per
month
    schedule_function(handle_data,
date_rules.month_start(), time_rules.market_close())
```

```
def month_perf(ts):
    perf = (ts[-1] / ts[0]) - 1
```

```
    return perf

def handle_data(context, data):
    # Get history for all the stocks.
    hist = data.history(context.universe, "close",
context.history_window, "1d")

    # This creates a table of percent returns, in order.
    perf_table =
hist.apply(month_perf).sort_values(ascending=False)

    # Make buy list of the top N stocks
    buy_list = perf_table[:context.stocks_to_hold]

    # The rest will not be held.
    the_rest = perf_table[context.stocks_to_hold:]

    # Place target buy orders for top N stocks.
    for stock, perf in buy_list.iteritems():
        stock_weight = 1 / context.stocks_to_hold

        # Place order
        if data.can_trade(stock):
            order_target_percent(stock, stock_weight)

    # Make sure we are flat the rest.
    for stock, perf in the_rest.iteritems():
        # Place order
        if data.can_trade(stock):
            order_target_percent(stock, 0.0)

def analyze(context, perf):
```

```
    # Use PyFolio to generate a performance report
    returns, positions, transactions =
pf.utils.extract_rets_pos_txn_from_zipline(perf)
    pf.create_returns_tear_sheet(returns,
benchmark_rets=None)
```

```
# Set start and end date
start = datetime(2003, 1, 1, tzinfo=pytz.UTC)
end = datetime(2017, 12, 31, tzinfo=pytz.UTC)

# Fire off the backtest
result = run_algorithm(start=start, end=end,
                       initialize=initialize,
                       analyze=analyze,
                       capital_base=10000,
                       data_frequency = 'daily',
bundle='quandl' )
```

As you can see in this backtest code, we trade only once per month and we use a new way to define that. Look at the last line of code in the initializ e function. There we set a scheduler, so that the trading code in the function handle_dat a will be run at the start of every month. This is an easy and convenient way of setting your trading frequency, and it can be used for other purposes as well.

In the initializ e function, we first define a list of stock tickers in the DJI. We

then make a list of corresponding symbol objects, and store in the context. A symbol object is a Zipline concept, which represents a particular stock.

Later in the handle_dat a function, you can see how we use a single line of code to pull data for all these stocks at once. This is much faster, not to mention much cleaner and more convenient than looping through them as we did earlier.

```
hist = data.history(context.universe, "close",
context.history_window, "1d")
```

The next step in the trading logic really shows off the beauty of Python. Pay attention to this one. It's a really useful trick.

What we are doing is to calculate a ranking table based on percentage returns for our 30 stocks. The code row in question is this.

```
perf_table =
hist.apply(month_perf).sort_values(ascending=False)
```

Even if you are familiar with other programming languages, this line may seem confusing to you. It may be a novel concept to many, but it's fairly easy to

understand and use.

From the code row before this, we have created the **DataFrame** object his t , by requesting historical closing prices for the stocks in the Dow Jones Index. Now we apply a function on this set of data.

The function in question simply calculates the percentage return between the first and the last data point. As we requested 20 trading days' worth of history, we get the performance for about a month back. The function, month_per f , was defined in the code as shown just below this paragraph. As you see, we just take the last, i.e. latest data point, divide it by the first data point and deduct one. Or in plain English, we just check the percent difference.

```
def month_perf(ts):
    perf = (ts[-1] / ts[0]) - 1
    return perf
```

What happens when we apply the function on the **DataFrame** is really clever. The percentage return will now be calculated for each stock, and a table will be returned with each stock and the

percentage return. In just one line of code, we applied this function and got the table back.

Since we are interested in the order of the percentage returns, we should also sort the data. That's what the last part of this row does, the sort_values( ) part. As you can see here, I supply the argument ascending = False. Since ascending sorting would be the default, if we don't supply this argument we would get the worst performers on top. Now we have a ready-to-use ranking table, and it would look something like Table 8.1, with symbols and percentage returns.

Table 8.1 Ranking Table

Equity(14576 [JPM])	0.044761
Equity(25942 [UNH])	0.031659
Equity(2482 [CAT])	0.021828
Equity(5213 [PG])	0.020452
Equity(3256 [XOM])	0.020132
Equity(2204 [PFE])	0.019902
Equity(3056 [WBA])	0.018613
Equity(2975 [UTX])	0.010529

Equity(6		5 [VZ])

In this example, we applied a very simple percentage return function on our data. This is meant to demonstrate the principle, and later on we are going to use this the same logic to perform more complex and more useful calculations.

Remember our stated rules; that we are to buy the top 10 performing stocks. Well, that should be easy at this point. Just slice the data, with the usual syntax of object[start:stop:step] . In this case as below. Note that we don't have to explicitly state the zero in front of the colon, nor do we have to specify step if we are fine with leaving it at the default value of 1.

```
buy_list = perf_table[:context.stocks_to_hold]
```

Now we know which stocks to hold and which not to hold. All we have to do now is to go over the stocks in the buy list and set their target weight to 10%, and to set the target weight of the rest to 0%.

That's the entire model code. That's all we need to get the performance data

we need, and in the next section we will look at how to analyze it.

Analyzing Performance with PyFolio

Creating a **PyFolio** analysis of the results requires surprisingly little code. In fact, it was already added to the code of the sample model in the previous section. It's so little code required that you might not even have noticed that I already put it in there.

The entire code to get a **PyFolio** report consist of importing the library, extracting the relevant data, and requesting a report. Three lines. We do need to have the **PyFolio** library itself installed in our `zip3 5` environment of course, as explained earlier in this chapter.

First up, we need to import the library in this particular piece of code. As

usual, we do that at the beginning of the code, as you have seen many times by now.

```
import pyfolio as pf
```

Now we just need wait for the backtest to complete, and for the analyz e function to kick in. You have seen in previous examples how this function is automatically run after a backtest, if it was specified in run_algorith m part.

Lucky for us, the **PyFolio** library is built to be used with Zipline. It can actually be used with other backtesting engines as well, but it's particularly easy to use with Zipline.

The second line of code needed for a **PyFolio** report is about extracting the information that we need from the backtest results. Specifically the returns, positions and transactions, using a **PyFolio** utility made for that purpose.

```
returns, positions, transactions =
pf.utils.extract_rets_pos_txn_from_zipline(perf)
```

In that code above, you see another handy feature of Python. A function can return more than one variable, which is

not the case with most other common languages.

```
def analyze(context, perf):
    returns, positions, transactions =
pf.utils.extract_rets_pos_txn_from_zipline(perf)
    pf.create_returns_tear_sheet(returns,
benchmark_rets=None)
```

PyFolio has various types of built in tear sheets, meant to show certain aspects of the results. Go ahead and explore the various features included on your own, but for now we will use the returns tear sheet.

I'll save you the hassle of going back to check the code of the complete model, and give it to you here again.

```
# Import a few libraries we need
from zipline import run_algorithm
from zipline.api import order_target_percent, symbol,
\
    schedule_function, date_rules, time_rules
from datetime import datetime
import pytz
import pyfolio as pf

def initialize(context):
    # Which stocks to trade
    dji = [
        "AAPL",
        "AXP",
```

```
    "BA",
    "CAT",
    "CSCO",
    "CVX",
    "DIS",
    "DWDP",
    "GS",
    "HD",
    "IBM",
    "INTC",
    "JNJ",
    "JPM",
    "KO",
    "MCD",
    "MMM",
    "MRK",
    "MSFT",
    "NKE",
    "PFE",
    "PG",
    "TRV",
    "UNH",
    "UTX",
    "V",
    "VZ",
    "WBA",
    "WMT",
    "XOM",
]
```

```
# Make symbol list from tickers
context.universe = [symbol(s) for s in dji]
```

```
    # History window
    context.history_window = 20
```

```
    # Size of our portfolio
    context.stocks_to_hold = 10
```

```
    # Schedule the daily trading routine for once per
month
    schedule_function(handle_data,
date_rules.month_start(), time_rules.market_close())
```

```
def month_perf(ts):
    perf = (ts[-1] / ts[0]) - 1
    return perf
```

```
def handle_data(context, data):
    # Get history for all the stocks.
    hist = data.history(context.universe, "close",
context.history_window, "1d")
```

```
    # This creates a table of percent returns, in order.
    perf_table =
hist.apply(month_perf).sort_values(ascending=False)

    # Make buy list of the top N stocks
    buy_list = perf_table[:context.stocks_to_hold]
```

```
    # The rest will not be held.
    the_rest = perf_table[context.stocks_to_hold:]
```

```
    # Place target buy orders for top N stocks.
```

```
    for stock, perf in buy_list.iteritems():
        stock_weight = 1 / context.stocks_to_hold

        # Place order
        if data.can_trade(stock):
            order_target_percent(stock, stock_weight)

    # Make sure we are flat the rest.
    for stock, perf in the_rest.iteritems():
        # Place order
        if data.can_trade(stock):
            order_target_percent(stock, 0.0)
```

```
def analyze(context, perf):
    # Use PyFolio to generate a performance report
    returns, positions, transactions =
pf.utils.extract_rets_pos_txn_from_zipline(perf)
    pf.create_returns_tear_sheet(returns,
benchmark_rets=None)
```

```
# Set start and end date
start = datetime(2003, 1, 1, tzinfo=pytz.UTC)
end = datetime(2017, 12, 31, tzinfo=pytz.UTC)

# Fire off the backtest
result = run_algorithm(start=start, end=end,
                       initialize=initialize,
                       analyze=analyze,
                       capital_base=10000,
                       data_frequency = 'daily',
bundle='quandl' )
```

When you execute this Python

backtest in **Jupyter Notebook** , you should get quite a nice display of information coming back to you. It should first show you some overview information and statistics, and then some useful graphs.

Table 8.2 PyFolio Key Ratios

Annual return	9.60%
Cumulative returns	295.20%
Annual volatility	19.20%
Sharpe ratio	0.61
Calmar ratio	0.17
Stability	0.78
Max drawdown	-58.20%
Omega ratio	1.12
Sortino ratio	0.86
Skew	0.19
Kurtosis	13.14
Tail ratio	0.97
Daily value at risk	-2.30%

If you are not familiar with all the analytics in Table 8.2, you may want to read up a little about them. It could very well be that some of them are not overly

interesting to you and your approach to the markets but it does not hurt to know more about such things.

For a quick overview of a strategy's performance, there are a few that you probably want to look at right away. The annualized return is the first thing most people would look at, though it tells only a small part of the story. High returns are generally better than low returns, but return alone is a useless number without context.

The annualized volatility as well as Sharpe ratio and maximum drawdown helps to put the annualized return figure into some context. These figures take volatility and downside risk into account.

What you should be looking for here are good numbers, but more importantly realistic numbers. All too often, I see how retail traders aim for fantasy numbers only to crash and burn when reality comes knocking. If your backtest numbers look too good to be true, they almost certainly are not true.

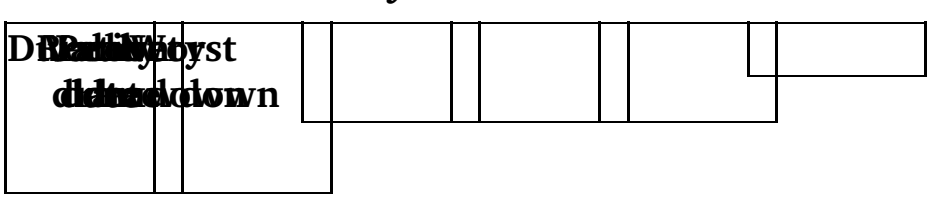
In real life, you are unlikely to

compound over 15% per year over any longer period of time. You are unlikely to achieve a Sharpe ratio of over 1 and you will probably see a maximum drawdown of three times your long term annualized return. These are very broad guidelines of course. Perhaps you can do a little better, perhaps not.

But if your backtest shows you annualized returns of 50%, with a maximum drawdown of 5% and a Sharpe ratio of 10, you have a problem with the backtest. Those are not achievable numbers in the real world.

The key ratios are only the first part of this return tear sheet. Next we find a drawdown table, which shows the top five drawdown periods, their percent loss, dates and recovery time, as shown here in Table 8.3. Again, this is displayed in my local date format, yyyy-mm-d d , and would probably look different on your own screen.

Table 8.3 PyFolio Drawdown Periods



	periods	
2004-08-30	2.17	0
2015-08-25	2.98	1
2004-08-30	1.09	2
2007-08-29	9.16	3
2003-08-21	0.00	4

The returns tear sheet also outputs quite a few graphs, designed to give you an overview of how the strategy behaves over time. A few of these graphs are shown below, to give you an idea of what this library will auto generate for you, and to let you see how this momentum strategy on the Dow performed.

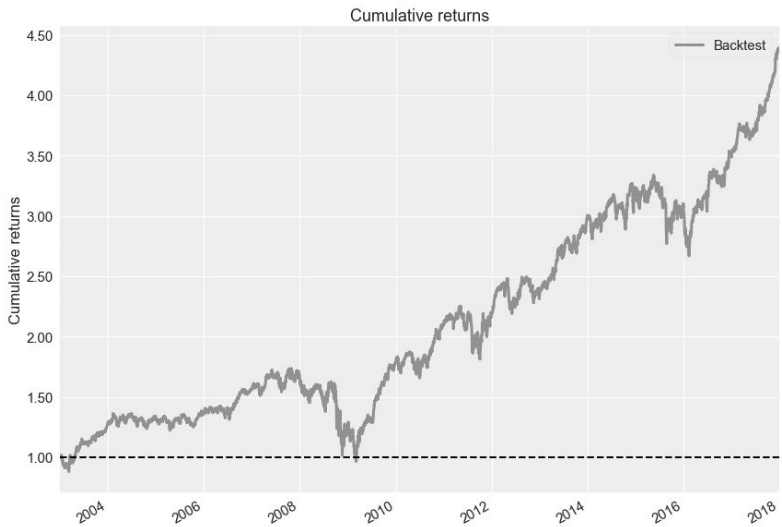


Figure 8-1 Pyfolio Cumulative Returns

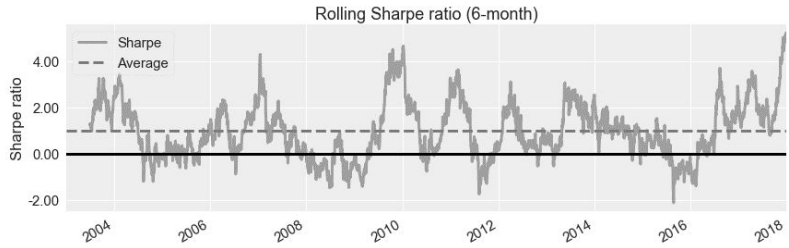


Figure 8-2 PyFolio Rolling Sharpe

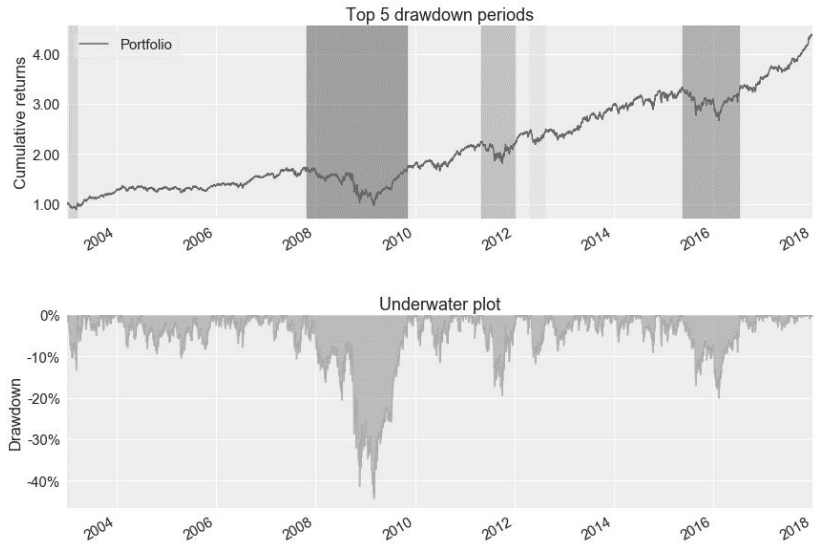


Figure 8-3 PyFolio Drawdowns

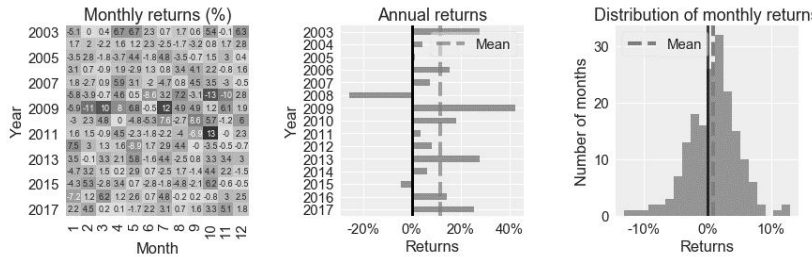


Figure 8-4 PyFolio Monthly and Yearly Returns

As you see, the **PyFolio** library can

generate a standard report which for many readers is probably all you need most of the time. A few lines of code, and you will get this report automatically. This should save you quite a bit of time.

Custom Analysis

You could also get all sorts of custom analysis. This is after all the beauty of Python, and the Pandas library. It makes it very simple to extract, manipulate and analyze time series data.

If you look back again at the code for the sample model that we are using for analysis here, you'll see that in the final row we store the results in a variable which we called simply `result` . Take a quick look at what this variable actually stores. This variable, `result` , is actually a **Pandas DataFrame** , and that greatly simplifies analysis. You can see that row again just here below.

```
# Fire off the backtest
result = run_algorithm(
    start=start,
    end=end,
```

```
    initialize=initialize,
    analyze=analyze,
    capital_base=10000,
    data_frequency = 'daily',
    bundle='quandl'
)
```

Assuming you just ran the model above in **Jupyter Notebook** , you can now continue to analyze simply by making a new cell below. Click the plus sign in the toolbar, and you get a new cell where you can write code, below the previous one.

You can run each cell separately and they can access the variables created in the previous cell. This way, you don't have to re-run the backtest, but can go right on to analyzing the result of the test you already did.

First you might want to check what kind of columns this **DataFrame** has. This can be done with this simple code.

```
for column in result:
    print(column)
```

That will print out the names of all the columns you have in the **DataFrame** , and your output will looks something like

this.

```
algo_volatility
algorithm_period_return
alpha
benchmark_period_return
benchmark_volatility
beta
capital_used
ending_cash
ending_exposure
ending_value
excess_return
gross_leverage
long_exposure
long_value
longs_count
max_drawdown
max_leverage
net_leverage
orders
period_close
period_label
period_open
pnl
portfolio_value
positions
returns
sharpe
short_exposure
short_value
shorts_count
sortino
starting_cash
starting_exposure
```

```
starting_value
trading_days
transactions
treasury_period_return
```

This will give you some idea of what is there, and how to work with it. But go ahead and go one step further. This **DataFrame** will contain one row for every day of the backtest run. That means that we can pick any individual day and check the state of the backtest on that particular day. We can output the values for an individual day easily, and even if this way of displaying the result is not very helpful for analytics, it does give us a clue to what is possible.

You can use .lo c to locate something inside a **DataFrame** based on simple criteria as shown below, or more complex logic once you get accustomed to it.

```
result.loc['2010-11-17']
```

That would output the field values, and you should see this kind of result.

```
algo_volatility
0.218396
algorithm_period_return
0.983227
```

alpha
2.63224
benchmark_period_return
1.4927e+08
benchmark_volatility
0.0132004
beta
-1.05027
capital_used 0
ending_cash
141.915
ending_exposure
19690.4
ending_value
19690.4
excess_return 0
gross_leverage
0.992844
long_exposure
19690.4
long_value
19690.4
longs_count
10
max_drawdown
-0.443695
max_leverage
1.00518
net_leverage
0.992844
orders []
period_close 2010-11-17
21:00:00+00:00
period_label

```
2010-11
period_open                                  2010-11-17
14:31:00+00:00
pnl                                                -96.598
portfolio_value
19832.3
positions                  [{'sid': Equity(290 [AXP]),
'amount': 48, 'las...
returns
-0.00484714
sharpe
0.50684
short_exposure
0
short_value                                              0
shorts_count                                             0
sortino
0.743699
starting_cash
141.915
starting_exposure
19786.9
starting_value
19786.9
trading_days
1985
transactions                                            []
treasury_period_return
0
Name: 2010-11-17 00:00:00+00:00, dtype: object
```

Day Snapshot

Looking at just an equity curve, the portfolio value development over the course of the backtest, can often create more questions than it answers. You may see some strange moves up or down, and now you wonder what could have caused this. The equity curve won't tell you, but you can always inspect the details of the days in question. That's usually a good way to check if the model is behaving the way it should, if it's trading the way you would expect, and holding the kind of positions you would expect.

Make a new cell in the Notebook below the previous one and try the following code.

```
# Let's get a portfolio snapshot

# Import pandas and matplotlib
import pandas as pd
import matplotlib.pyplot as plt

# Select day to view
day = '2009-03-17'

# Get portfolio value and positions for this day
port_value = result.loc[day,'portfolio_value']
day_positions = result.loc[day,'positions']
```

```
# Empty DataFrame to store values
df = pd.DataFrame(columns=['value', 'pnl'])

# Populate DataFrame with position info
for pos in day_positions:
    ticker = pos['sid'].symbol
    df.loc[ticker,'value'] = pos['amount'] *
pos['last_sale_price']
    df.loc[ticker,'pnl'] = df.loc[ticker,'value'] -
(pos['amount'] * pos['cost_basis'])
```

```
# Add cash position
df.loc['cash', ['value','pnl']] = [(port_value -
df['value'].sum()), 0]

# Make pie chart for allocations
fig, ax1 = plt.subplots(figsize=[12, 10])
ax1.pie(df['value'], labels=df.index, shadow=True,
startangle=90)
ax1.axis('equal')
ax1.set_title('Allocation on {}'.format(day))
plt.show()

# Make bar chart for open PnL
fig, ax1 = plt.subplots(figsize=[12, 10])
pnl_df = df.drop('cash')
ax1.barh( pnl_df.index, pnl_df['pnl'], align='center',
color='green', ecolor='black')
ax1.set_title('Open PnL on {}'.format(day))
plt.show()
```

Here we pick a date at the top, where we want to see the current allocation and open position profit and

loss. At first, we use the helpful .loc to locate the matching date. With this tool, we can find a specific column value for a specific row, with this syntax .loc[row, column] . Note that this simple code has no error handling yet, in case you pick a date that's not part of the result object, such as a weekend.

Based on the row that matches the date we gave, we can now find the info needed to make some graphs. As you saw earlier, there is a row in the result object which holds a list of positions. What this code does it to iterate through those positions and make a **DataFrame** with the value and the open profit and loss. This could be done more elegantly, but this book is about clarity and not elegance.

For an allocation pie chart to make sense, we also need to add any cash position. We are dealing with equities here, so the cash holdings would simply be the total portfolio value minus the value of the stocks we hold. After we have added this, we can draw a nice pie using the **MatPlotLib** , just as we have done earlier in this book.

Making a horizontal bar graph for the open profit and loss per position is just as easy, and that's what the last section of the code sample does.

This is just meant to demonstrate the relative ease of zooming in on a particular day and getting a feel for how the portfolio looked at that time.

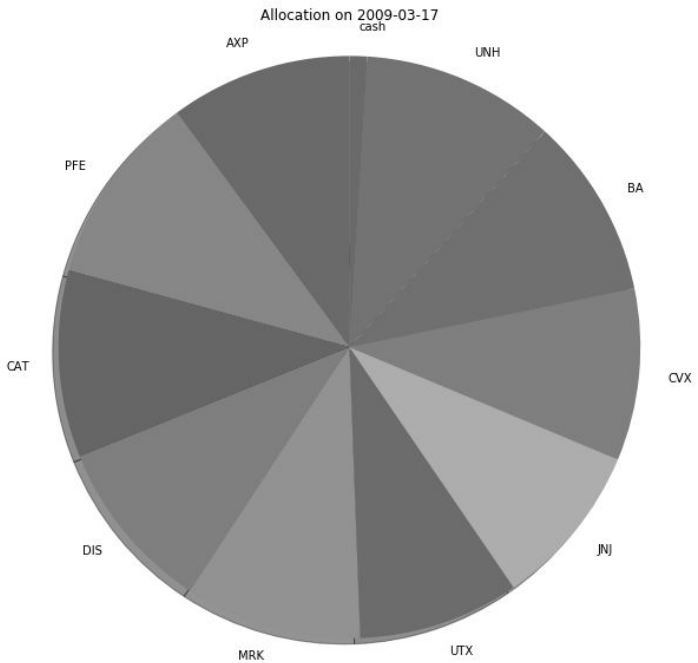


Figure 8-5 Day Snapshot Allocation

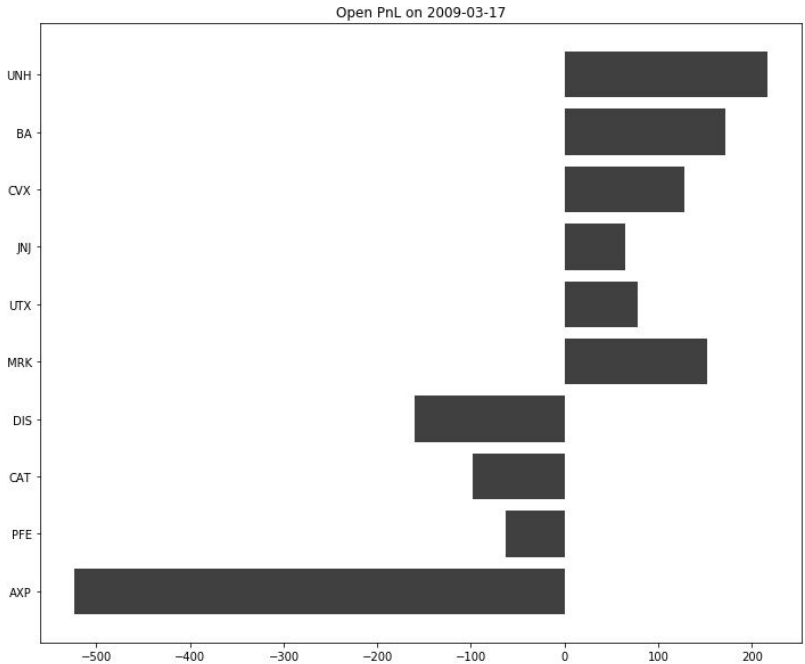


Figure 8-6 Day Snapshot Open Profit and Loss

Custom Time Series Analytics

Time series analytics is where Python really shines. The only thing stopping you here is your own creativity. The more you work with backtesting the more you will find yourself wondering about various aspects of the return curves that comes out on the other end.

To give you an idea of how quickly and easily you can calculate various analytics, I will show an example here. We are going to make a graph with four sub plots. The first will show a semi-log graph of the equity curve itself. The second will show the exposure held over time. So far, those are time series that we already have. But the next two we need to calculate.

The third plot will show half a year rolling returns, on an annualized basis. I have chosen this analytic because it makes for a good learning exercise. That means that we need to stop a moment to think about what this is.

We all know, or least we should all know, what annualized return refers to. Normally, you would probably look at the annualized return over the course of the entire backtest. You may at times see the term Compound Annual Growth Rate, or CAGR, which is the same thing.

If for instance, you started out with $10,000 and managed to achieve an annualized return of 10% for ten years,

you would end up with $25,937.

$$(1 + 0.1)^{10} \times 10,000 = 25,937$$

Or to turn it around, if you started out with $10,000 and ended up with $25,937 after ten years, you could easily find the annualized rate of return.

$$\left(\frac{25,937}{10,000}\right)^{1/10} - 1 = 10\%$$

But such a number may move up and down a lot during a backtest, and it can be useful to see what the annualized return would have been on a shorter time frame, on a rolling basis.

This is very basic financial math and very simple when you are dealing in full years. It can be a very useful tool to be able to calculate annualized numbers on shorter or longer time periods than a full year, and that's a more realistic scenario. In the real world, you will almost always be dealing with such time series.

Let me therefore give you a more generic formula for dealing with this. I will give you the formula first, and then you will see in the code below how easily

we apply this math to a Python series.

$$\text{AnnualizedReturn} = \left(\frac{\text{EndValue}}{\text{StartValue}}\right)^{\frac{\text{YearDays}}{\text{ActualDays}}} - 1$$

Using this simple math, you can quickly find the annualized return, whether time period in question is longer or shorter than a year. Now look at how we can make a generic function to calculate this.

```
def ann_ret(ts):
    return np.power((ts[-1] / ts[0]), (year_length/
len(ts))) -1
```

In this function, we feed in a time series, and get an annualized return rate back. The only thing we need to define is just how many days we are assuming in a full year. As you will see in my code, I tend to assume 252 trading days, which is pretty close to most years' actual days. Some prefer 256 days, as it makes can calculations easier.

The reason that I'm defining a separate function for this is that it makes it so much simpler to apply it to a rolling time window, as we will soon see in the code.

For the same reason, I'm making a

function to calculate the maximum drawdown on a time window as well.

```
def dd(ts):
    return np.min(ts / np.maximum.accumulate(ts)) - 1
```

Here we provide a time series window, and get back the value for the deepest drawdown during this period.

The rest of the code is mostly about formatting a nice looking graph. I will assume here that you are working in **Jupyter** , and that you have just run the model in the previous section. Any model will do, actually. Now make a new cell just below, where we will write the new code.

As always, you can find this code as well at the book website, **www.followingthetrend.com/trading-evolved/** .

Here is the code we need to calculate the mentioned analytics.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import matplotlib.ticker as ticker
```

```
# Format for book image
font = {'family' : 'eurostile',
        'weight' : 'normal',
        'size'   : 16}
matplotlib.rc('font', **font)

# Settings
window = 126
year_length = 252

# Copy the columsn we need
df = result.copy().filter(items=['portfolio_value',
'gross_leverage'])

# Function for annualized return
def ann_ret(ts):
    return np.power((ts[-1] / ts[0]), (year_length/
len(ts))) -1

# Function for drawdown
def dd(ts):
    return np.min(ts / np.maximum.accumulate(ts)) - 1

# Get a rolling window
rolling_window =
result.portfolio_value.rolling(window)

# Calculate rolling analytics
df['annualized'] = rolling_window.apply(ann_ret)
df['drawdown'] = rolling_window.apply(dd)

# Drop initial n/a values
df.dropna(inplace=True)
```

After the required import

statements, you see some odd looking rows about font. I have included this just to show you how I formatted the graphs to be displayed in this book. You can safely skip this if you like, it's just left there in case you are curious about fond and size formatting.

```
# Format for book image
font = {'family' : 'eurostile',
        'weight' : 'normal',
        'size'   : 16}
rc('font', **font)
```

Then there are two settings. The window setting defines what length of rolling time window we want to analyze. The number in the code, 126 days, represents approximately half a year. Change it to anything you like.

The second setting is for how many business days a year we are assuming, and this should be approximately 252 days.

```
# Settings
calc_window = 126
year_length = 252
```

Then we construct a **DataFrame** , simply by copying the only two columns that we need from the **DataFrame** that

was created by the backtest earlier. Remember that we made a simple backtest earlier in this chapter, which returned a variable called resul t . We have no need for most of what is in this variable, so we just copy portfolio_valu e and gross_leverag e into a new **DataFrame** .

```
# Copy the columsn we need
df = result.copy().filter(items=['portfolio_value',
'gross_leverage'])
```

You will find two functions in the code, one to calculate annualized return and one to calculate drawdown.

```
# Function for annualized return
def ann_ret(ts):
    return np.power((ts[-1] / ts[0]), (year_length/
len(ts))) -1

# Function for drawdown
def dd(ts):
    return np.min(ts / np.maximum.accumulate(ts)) - 1
```

Follow the code down a bit from there, and you see how we are defining a rolling time window of data, and using the same apply logic as we have seen previously to get rolling time series of both analytics.

```
# Get a rolling window
```

```
rolling_window =
result.portfolio_value.rolling(calc_window)

# Calculate rolling analytics
df['annualized'] = rolling_window.apply(ann_ret)
df['drawdown'] = rolling_window.apply(dd)
```

Now we have a **DataFrame** with everything calculated for us, and all we need to do is to visualize it.

```
# Make a figure
fig = plt.figure(figsize=(12, 12))

# Make the base lower, just to make the graph easier
to read
df['portfolio_value'] /= 100

# First chart
ax = fig.add_subplot(411)
ax.set_title('Strategy Results')
ax.plot(df['portfolio_value'],
        linestyle='-',
        color='black',
        label='Equity Curve', linewidth=3.0)

# Set log scale
ax.set_yscale('log')

# Make the axis look nicer
ax.yaxis.set_ticks(np.arange(df['portfolio_value'].min(),
df['portfolio_value'].max(), 500 ))
ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.0f'

# Add legend and grid
```

```
ax.legend()
ax.grid(False)

# Second chart
ax = fig.add_subplot(412)
ax.plot(df['gross_leverage'],
        label='Strategy exposure'.format(window),
        linestyle='-',
        color='black',
        linewidth=1.0)

# Make the axis look nicer
ax.yaxis.set_ticks(np.arange(df['gross_leverage'].min(),
df['gross_leverage'].max(), 0.02 ))
ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.2f')

# Add legend and grid
ax.legend()
ax.grid(True)

# Third chart
ax = fig.add_subplot(413)
ax.plot(df['annualized'],
        label='{} days annualized
return'.format(window),
        linestyle='-',
        color='black',
        linewidth=1.0)

# Make the axis look nicer
ax.yaxis.set_ticks(np.arange(df['annualized'].min(),
df['annualized'].max(), 0.5 ))
ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f')
```

```
# Add legend and grid
ax.legend()
ax.grid(True)

# Fourth chart
ax = fig.add_subplot(414)
ax.plot(df['drawdown'],
        label='{} days max drawdown'.format(window),
        linestyle='-',
        color='black',
        linewidth=1.0)

# Make the axis look nicer
ax.yaxis.set_ticks(np.arange(df['drawdown'].min(),
df['drawdown'].max(), 0.1 ))
ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f')

# Add legend and grid
ax.legend()
ax.grid(True)
```

The output of this should be four graphs, similar to what you see in Figure 8-7. As you start getting comfortable with constructing backtesting models in Python, you will probably find yourself creating all kinds of custom analytics and graphs to visualize what is important to you, to locate issues or learn more about the behavior of the models you have built.

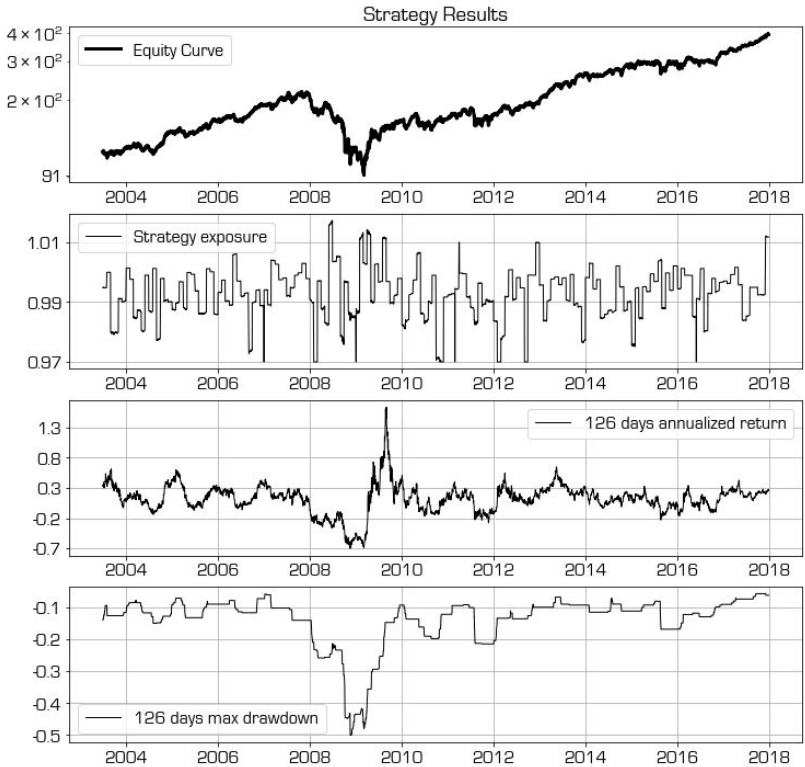


Figure 8-7 Custom Time Series

9 Exchange Traded Funds

Exchange traded funds, or ETFs, are one of those great finance ideas which were eventually turned into a way to sucker people out of their hard earned cash. Say what you will about bankers,

but they are very impressive when it comes to finding creative ways of getting to your hard earned cash.

Many ETFs are great. But you shouldn't assume that some fund is great just because it's packaged as an ETF. Anybody can make an ETF out of almost anything. The term itself has become nearly meaningless.

Roughly speaking, ETFs can be divided into three buckets.

The Good

The best ETFs are passive, low cost index trackers. Transparent, highly liquid instruments which do exactly what their name states. The best example would be the world's most liquid ETF: the SPY. This fund invests its assets as close to the S&P 500 index as possible, using a computerized method to ensure low cost and close tracking. When you buy something like the SPY, you know exactly what you get. You get the index, minus about 10 basis points per year in cost.

That's a whole lot better than what the mutual fund industry can offer you.

Roughly 80 to 90% of all mutual funds fail to beat their benchmark. That's widely known these days, but for some reason people still invest in such funds. The primary reason is likely that the banks are advising their clients to buy their own overpriced and underperforming mutual funds. Rather than rolling your dice on a 10 to 20% probability of outperforming the index, you can just buy a low cost index tracker and know exactly what you get.

The SPY has at the moment about 225,000,000,000 dollars under management. Yes, that would be 225 billion bucks. Close to a quarter of a trillion. It's more than twice the size of the second largest ETF and has a very high trading volume. As far as ETFs go, it's really in a category by itself in terms of liquidity. I wish I could say that I get paid for peddling the SPY, since these guys are making more money than a medium sized European nation, but sadly I have yet to see any checks from them.

There are many other ETFs that operate according to the same principles and are highly liquid. These types of ETFs, the ones that passively track a clearly defined index at a low cost and narrow tracking error, are the primary target for systematic trading.

Such ETFs are easily found among those tracking major country indexes, sector indexes and even some bond indexes. The very liquid ones are usually possible to short, while the rest will be very hard to locate in any decent scale.

This is what ETFs were to begin with and that's what most people would associate the term with. What you need to be careful with though, is with ETFs that don't fit the index tracking low cost design.

The Bad

Anybody can turn anything into an ETF. Think about that for a moment. The term does not mean anything except that it's a fund that can be bought and sold via

some sort of exchange. It does not mean liquid, it does not mean low cost and it does not mean index tracking. It does not even mean that it must be transparent.

Some ETFs are extremely small and extremely illiquid. There are ETFs out there with assets of less than 5 million. In the ETF space, anything less than a billion is considered tiny. Anything under 100 million is just silly. If you trade anything that small, be very careful with executions and don't trade too big.

Check the net asset value of an ETF before considering it, and check the daily trading volume. You want to make sure that you can easily go in and out of the fund. If your position is 10% or more of the average daily trading volume, you are far too exposed.

Another issue is that many of the instruments that we normally call ETFs, are not actually ETFs. They are ETNs, Exchange Traded Notes. It may sound close enough, but it's really not.

With an ETF, you theoretically own the underlying shares. The ETF holds a

basket of securities for you. If the ETF provider blows up, you should be able to get your part of the underlying shares back. If you have a large enough position, usually 50,000 shares, you can actually ask for the shares at any time.

ETNs on the other hand are in effect structured products. They are debt instruments, where a company promises to pay you based on the performance of an underlying index. That index may be a stock market or it may be absolutely any sort of mathematical formula at all. You don't hold any actual claim to any underlying security, you just hold that promise. How the ETN provider hedges their side to be able to pay you is their business. If they go the way of the dodo, you are just out of luck. You won't be seeing that cash again.

If you think that it's an unlikely prospect for a large bank or financial service provider to take a sudden dirt nap and leave you out to dry, you probably were not around in the business in 2008. This has happened before and this will happen again. Be careful with ETNs.

Then there is the cost side. The term ETF by its very nature conjures an image of low cost and responsibility. But again, anyone can make anything into an ETF. And set their own terms. There is nothing stopping me from setting up a new ETF and charging a two percent management fee. It's probably not going to be an easy sell, but very much possible.

The ETN structure is common when it comes to commodity funds. With such funds, you need to look closely at what they actually are and what they actually do. For a standard index ETF, the fund can just buy the underlying. But you shouldn't expect that a crude oil fund will match the return of physical crude. Think about it. How would that happen? Would the provider go buy the barrels? And put where? How?

No, they would of course buy and roll oil futures. That means that if you want to understand your return profile for an oil ETN, you need to understand futures and the impact that term structure has over time. But you can't be sure that the vehicle will invest only in what its

name would imply.

Take the USO for instance, which is really not a bad tracker. As of writing this, it has about 3 billion net assets and a pretty high daily turnover. This should be a great way to track oil, right? Well, that all depends on your definition of great and what you would expect.

A quick look at this fund tells us that they do indeed invest 100% in oil. As of writing this, the USO has exactly 100.0% exposure to March 2017 crude oil futures. Yes, the futures. Of course. As mentioned, they couldn't very well be expected to keep actual barrels of oil in the basement.

But their investments don't stop there. On top of the 100% in crude futures, there are a few more things in the portfolio. First there is 10.4% in Fidelity Institutional Government Portfolio, 7% in the equivalent Goldman product, another 7 in Morgan Stanley's money market product and so on. Why? Well, because they have plenty of free cash. The futures don't require much cash, so rather than

keeping this under the mattress, any futures manager would make use of short term money market products to secure the money and hopefully get a small return on it.

This in itself is not really an issue. But I mention is so that you understand that these commodity ETF/ETN products are akin to futures funds. They have little in common with index trackers.

The issue here is that retail investors might see the name and the description of such a fund, and expect to get a return similar to the underlying spot. That's not likely to happen. Readers familiar with futures and term structures already know why.

We are going to look closer at term structure later on in the futures section, so I won't go too deeply into that for the moment. What is important to understand is that futures are different than spot. The time factor of the futures instruments will over time create a very different return curve than the underlying. That's not the fault of the fund manager. That's reality.

You could perhaps fault them for not making such things clearer to investors, but the effect itself is quite unavoidable.

The stated objective of the USO fund is this: “The fund seeks to reflect the performance of the spot price of West Texas Intermediate light, sweet crude oil delivered to Cushing, Oklahoma by investing in a mix of Oil Futures Contracts and Other Oil Interests”. The fund was launched in April 2006. Let’s see how that objective worked out for them since then.

Figure 9-1 USO ETF vs. Spot Oil

As the chart shows, the tracking is not exactly close. But that’s as close as

you will get. That does not mean that you should never trade a commodity ETF or ETN, but you do need to be aware of what you are actually buying. The name of a fund is not enough to tell you what is really going on inside it.

The easiest way to check how a fund compares to the underlying benchmark is to just pull up a chart. Since my hidden agenda with this book is to teach you some Python when you least expect it, I will throw a few lines of code at you.

First I checked the product page for the USO to verify which exact benchmark it uses. I then went about locating the data for this benchmark, the West Texas Intermediate, as well as the equivalent data series for the ETF. You should be able to find both for free online, but as free sources keep changing location or suddenly go premium, I took the liberty of storing this data on **www.followingthetrend.com/trading-evolved/** where you can download it.

The next part is simple, if you paid

attention in earlier chapters of this book.

```
import pandas as pd
import matplotlib.pyplot as plt

# Read data from csv
df = pd.read_csv('oil_etf_vs_spot.csv', index_col='Date',
parse_dates=['Date'])

# Make new figure and set the size.
fig = plt.figure(figsize=(12, 8))

# The first subplot, planning for 3 plots high, 1 plot
wide, this being the first.
ax = fig.add_subplot(111)
ax.set_title('Oil ETF vs. Spot')
ax.plot(df['WTI-West-Texas-Intermediate'],
linestyle='-', label='Spot', linewidth=3.0,
color='black')
ax.plot(df['USO'], linestyle='--', label='ETF',
linewidth=3.0, color = 'grey')
ax.legend()
```

This code assumes that you have downloaded the aforementioned data from my site, and put in the same folder as your Python code. If you got the data elsewhere, or store it elsewhere, just adapt your code to read the corresponding format and location. Executing this code should result in the same output as Figure 9-1

While term structure of the futures market is a large factor at play here, another thing to watch out for it the cost. The SPY takes less than ten points, less than 0.1% management fee, but there are funds taking a percent or more. Many of those are active. The USO mentioned above has a total expense ratio, or TER of 0.74%.

I might be a little unfair to single out the USO here. It's really not that bad, as far as commodity ETFs go. Try the same comparison for other exchange traded commodity funds, and you will see that they are all the same, or sometimes noticeably worse.

Next we have active ETFs. Active funds are just what they sound like. Somebody is actually trading and making discretionary decisions. Most people would think it's not an ETF if it involves people sitting by the screens and doing manual trading, but it certainly can be. As long as it's exchange traded. If the strategy is active, the risk picture changes dramatically. You can no longer be sure of what you actually get.

Sometimes there is little choice but to trade the bad ETFs. These are funds that you should be very careful with, but properly understood there may still some use cases for them. Just make really sure that you understand what you are trading.

The Worst

The final segment of ETFs are the really horrible ones. I would have called the category the ugly, but I'd hate to tarnish a great film with these junk products. The structured products, all dressed up to be sold to retail traders who have no idea what structured products are. The worst offenders are the leveraged and inverse ETFs.

The short ETFs are the most obvious traps. The ProsShares Short S&P500 (SH) for instance, which promises you the inverse performance of the S&P 500 index. In all fairness, it pretty much does. But not in the way that most would expect. The key to understanding these products, because they really are

structured products, is that they have a daily rebalance. They are designed to produce approximately the inverse return on any single day. The only way to achieve that is a daily rebalance.

Whenever you deal with any sort of derivative, you always need to think in terms of hedging. If you know how to hedge a derivative, you know how to price it. If you understand that, you understand how the instrument works and what you can really expect.

The way to create an inverse ETF is by using futures or swaps. If the ETF has two billion in assets, which is about what the SH has, they need to put on a short exposure of two billion, using derivatives. This position is then rebalanced daily to ensure that the daily return of the ETF is inverse from the underlying market, any given day.

The effect of this is that you are short volatility. In options terminology, you thought you took a short delta position but ended up with a mostly short gamma position. But that probably sounds

like Greek to most readers.

Try an example instead. Assume we have an index which keeps wobbling up and down in a range for a long time. In this example, the index will be moving up and down by exactly the same amount every day, resulting in no long term price progress at all. On the first day, it moves up 3 percent. The second day, it drops 2.91%, putting it back where it started. Now it keeps doing that for some time. One might think that a long ETF on this index, a short ETF and a double short ETF would all show a flat return over time as well. But that's far from reality.

Table 9.1 shows the effect on a short ETF and a double short ETF over just a few days of such back and forth price flips. After seven days, the underlying index is back at exactly 100, while the short ETF is down half a percent, and the double short is down one and a half.

Table 9.1 Effect of Short ETF

Day	Underlying	Inverse	2x Inverse

| | | |

In this example, I wanted to show this phenomenon and how it develops over 50 days. Therefore, I start off by making a **DataFrame** with a numbered index, from zero to 49. Remember from earlier how we can get a range of numbers with the range( ) function.

```
df = pd.DataFrame(index=range(50))
```

Now let's create a new column for the daily percentage change of our theoretical underlying asset. As per Table 9.1, we want every second day to be +3% and every alternate to be -0.2913%, so create a net zero return over time. We find every second in the sample code below by checking the reminder if divided by two. Note how we use lo c again to locate the rows and columns we want to change.

```
# Set odd days to +3%
df.loc[df.index % 2 == 1,'underlying_return'] = 0.03

# Set even days to -2.913%
df.loc[df.index % 2 == 0,'underlying_return'] =
-0.02913
```

Now that we have the daily percent return numbers, we can easily calculate

the cumulative product. That is the performance over time, for the underlying, the inverse ETF and the double inverse ETF.

```
# Calculate cumulative series
df['underlying_price'] = (df['underlying_return'] +
1).cumprod()

# Inverse ETF
df['short_return'] = df['underlying_return'] * -1
df['double_short_return'] = df['underlying_return'] * -2
# Double Inverse
df['short_price'] = (df['short_return'] + 1).cumprod()
df['double_short_price'] = (df['double_short_return'] +
1).cumprod()
```

The rest of the code simply plots the graphs. Below you'll find the complete code used for this example.

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame(index=range(50))

# Set odd days to +3%
df.loc[df.index % 2 == 1,'underlying_return'] = 0.03

# Set even days to -2.913%
df.loc[df.index % 2 == 0,'underlying_return'] =
-0.02913
```

```
# Start at zero
df.iloc[0].loc['underlying_return'] = 0

# Calculate cumulative series
df['underlying_price'] = (df['underlying_return'] +
1).cumprod()

# Inverse ETF
df['short_return'] = df['underlying_return'] * -1
df['double_short_return'] = df['underlying_return'] * -2

# Double Inverse
df['short_price'] = (df['short_return'] + 1).cumprod()
df['double_short_price'] = (df['double_short_return'] +
1).cumprod()

# Make  new figure and set the size.
fig = plt.figure(figsize=(12, 8))

# The first subplot, planning for 3 plots high, 1 plot
wide, this being the first.
ax = fig.add_subplot(111)
ax.set_title('Short ETF Effect')
ax.plot(df['underlying_price'], linestyle='-',
label='Spot', linewidth=3.0, color='black')
ax.plot(df['short_price'], linestyle='-', label='Inverse
ETF', linewidth=3.0, color = 'grey')
ax.plot(df['double_short_price'], linestyle='--',
label='Double Inverse ETF', linewidth=3.0, color =
'grey')
ax.legend()
```

The output from this code would look as Figure 9-2. Once you understand

this fairly simple mathematical example, you should understand the problem with inverse ETFs, and why you should almost never trade them.

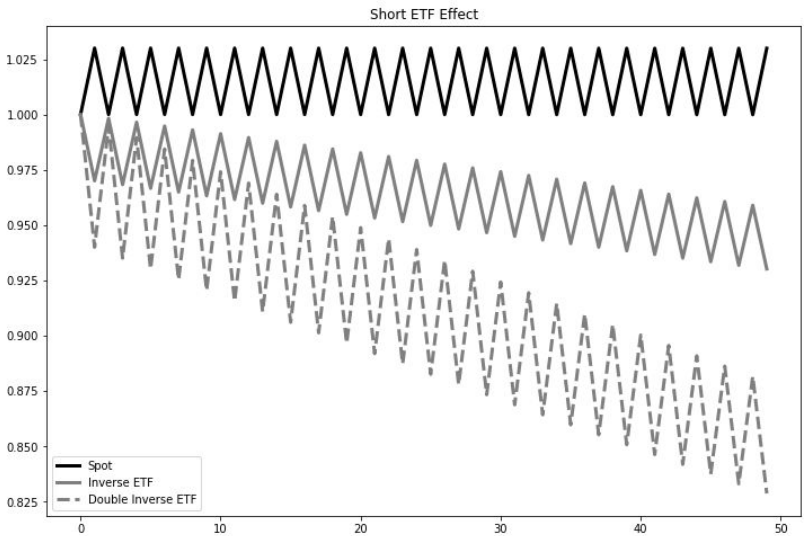


Figure 9-2 Effect of Short ETF

The obvious problem of course, is that when markets start falling, they also tend to get quite volatile. You may very well find that even during declining markets, the inverse ETFs often also lose value.

This is not an instrument for hedging. This is, at best, an instrument for very short term speculations. You should have a very good reason for ever holding an inverse ETF for more than a day.

The example above is highly theoretical, but you could easily test this on actual market data. You can download a data file from my site, www.followingthetrend.com/trading-evolved/, which contains price series for the index tracker SPY, the inverse ETF SH and the double inverse SDS, all with the same underlying S&P 500 Index.

If you are still unsure about short ETFs, download this data and compare performance from different starting points. Try starting in different type of markets regime, and see what the effect was over weeks and months.

Here is a simple piece of code to read the data file you can find on my site, recalculate the series to the same starting value, and plot a comparison graph.

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt

# Function for recalculating series with a common base
def rebased(ts):
    return ts / ts[0]
```

```
# Read data from csv
df = pd.read_csv('short_etfs.csv', index_col=='Date',
parse_dates=['Date'])

# Calculate all series starting from first value.
df = df.apply(rebased)

# Make  new figure and set the size.
fig = plt.figure(figsize=(12, 8))

# The first subplot, planning for 3 plots high, 1 plot
wide, this being the first.
ax = fig.add_subplot(111)
ax.set_title('Short ETF Effect')
ax.plot(df['SPY'], linestyle='-', label='SPY Index
Tracker', linewidth=3.0, color='black')
ax.plot(df['SH'], linestyle='-', label='SH Short ETF',
linewidth=3.0, color = 'grey')
ax.plot(df['SDS'], linestyle='--', label='SDS Double
Inverse ETF', linewidth=2.0, color = 'grey')
ax.legend()
```

This should produce a graph like the one you see in Figure 9-3. This is no longer a theoretical example. The performance you see in that figure shows the actual performance of the S&P index tracker and compared to the inverse and double inverse ETF. Yes, the inverse funds spike up during short term market distress, but as you can see, over the course of months and years, the index

tracker can lose substantially, with the inverse funds losing at the same time.

Don't touch the inverse trackers unless you fully understand this behavior.

Figure 9-3 Inverse S&P 500 ETFs

Shorting Exchange Traded Funds

Just because your backtester does not complain when you enter a simulated short sell, don't go assuming that it would be possible in reality.

The most liquid ETFs are generally

possible to locate and short. With others, it's very slim pickings. Building algorithmic trading models that short ETFs is a dangerous thing. It looks so easy in the backtester. Just enter the order, and there you go.

In reality, a safe assumption is that apart from the really enormously liquid ETFs, you won't be able to short. At least not at the time when you want to short. And of course, even if you get that lucky, the shares are likely to be recalled at the worst possible time.

Shorting is more complicated than it looks on the screens. It's easy to forget at times, since modern trading looks increasingly like a computer game, but there is actually something real happening behind the scenes.

When you short a stock or ETF, someone is actually lending it to you. This is important to understand. Your transaction does not happen in a vacuum. Understanding the mechanics behind shorting is the key to understanding what is realistic and what is not.

In most simulation platforms, you can short anything in any scale at zero cost. This is very far from reality.

First your broker needs to locate the shares. This means that they need to have another client who is happy to hold those shares long term and is willing to lend them for shorting. This usually means an institution who has a long term holding and wants to make a little extra on the side on the lending. You did not think you would get to borrow the shares for free, did you?

Borrowing major ETFs and stocks is usually no problem, even in decent scale. But if you want to short ETFs beyond the very top, massively liquid ones, you will likely find it quite difficult to locate them. Even if you do find them, they might be expensive.

Shorting the SPY is easy. It's just ridiculously liquid and extremely widely owned. You can borrow it reliably and at good rates. But if you want to short something more exotic, like a natural gas ETF for instance, you will either find that

there are no lenders available or that you will have to pay so much that it won't be worth it.

The lenders will need to be compensated. They are not lending you the shares because they are such kind human beings. That's rarely how finance works. You will be paying the equivalent of an interest rate on them. How high that is will depend on how difficult the share is to borrow and how important you are to your broker. If you plan to hold your shorts over days, or even weeks and months, this cost can add up and make a serious dent in your strategy. It's a factor that's very easy to forget about, and one that can potentially kill your strategy returns.

Another often forgotten part about short selling is that the shares can be called back at any time. Yes, even if your strategy is sound, the rug can be pulled from beneath your feet without notice.

It's fair to assume that as a short seller, you are at your most happy when stocks are falling hard. And that also

happens to be when the lender of the shares is feeling the most pain. So when is he likely to call back the shares?

You just can't model this factor. There is no historical data on when stocks are recalled, and you can't predict how that might develop in the future. Consider what is going on at the other side of the table, and how they might think. Don't assume that you can short something whenever you like, however you like, and hold for as long as you like. That's not how it works.

This can be easily forgotten when building trading models. Almost all backtesting software will let you short anything at any time, with no funding costs or regards for availability. It can be extremely difficult to properly model the short side of ETFs, or stocks for that matter.

10 Constructing ETF Models

Building ETF trading models is a great way to learn the basics. It's simple for several reasons. There are not many of them. That makes things much easier than for something like stocks. There are more stocks out there than anyone cares to count. Having a very large set of instruments to pick from has both advantages and disadvantages, but it clearly adds a level of complication. Even with the boom of the ETF business and new funds popping up all over, the extent of the tradable ETF universe is fairly limited.

ETFs are, much like stocks, cash instruments. That's another factor that makes things simple for us. In the financial world, cash instruments are the most simple. You generally pay up front for what you want to buy, you have a linear payoff and the percent change in the instrument is your profit or gain. Very easy, compared to the likes of futures, options, swaps, forwards and other derivatives.

Many ETFs, at least most of the good ones, are clear and transparent. They

usually track a specified index, so you know exactly what is in it and thereby you know what to expect from the returns. Naturally you don't know what to expect in terms of exact returns, but you can analyze the underlying index and gain an understanding of the volatility and return profile in various market climates.

There are downsides with trading ETFs as well, even if you stick to just the good ETFs and not the structured product kind. A common mistake is to think that ETFs can simply replace futures in creating diversified strategies. That's just not true. The asset class coverage just is not good enough to replace futures, but the cash nature of ETFs is the real killer. The point with diversified futures strategies is that you can trade anything, in any asset class, without worrying about cash constraints.

While futures traders can focus solely on the risk and completely disregard notional exposure, an ETF trader can't afford such luxury. When cash runs out, it runs out. Leveraging up from there is restrictive and expensive.

Most ETFs are fairly low volatility. Take something like a classic index tracker. If the underlying index consists of 500 stocks, the index tracker will naturally have a lower volatility than most constituent stocks. That in itself is not a problem, but in combination with the cash constraint it can be.

Taking on lots of volatility or a high notional exposure in a portfolio is not necessarily a good thing. But it's a good thing to have the option of doing so, if that's what you need.

In the futures world, there is no worry about notional exposure. A good example of when this way of working is useful is in the money markets. To be clear, money market refers to the short term interest rate sector. Not to the currency markets, which is a very different thing.

Money market instruments have a very low volatility, due to their short duration. At this point, it's not too important to understand why, just that in this sector a 0.1% daily moves is

considered large and 1% daily move is practically unheard of.

Trading such slow moving markets is not a problem in futures space. You just buy more of it. In futures world, the limit to how much you can buy is defined by how much risk you want to take, rather than how much cash is in your portfolio. In theory, the margin requirements set the limit for how much risk you can take with futures, but it's quite unlikely that you would like to take that much risk on. For ETFs though, the situation is very different.

Slow moving, low volatility ETFs, will hog a lot of cash for very little return. The cash availability is limited, and if you use it to buy an instrument that barely moves a few basis points a day, you are simply locking up capital. You just can't leverage up the same way as you could for futures.

Still, because of the relative simplicity of ETFs, it's a well suited area to start learning about constructing trading models.

Asset Allocation Model

It its simplest form, an asset allocation model is simply a mix of broad asset classes, held at a certain predetermined weight. This type of model is more of a long term investment approach than trading, but can make a lot of sense for capital allocation.

An age old allocation approach would be to place 70 percent of your money in bonds and 30 percent in stocks. That's the rule that Warren Buffet famously recommends for retirement savings. Many readers of this book will surely think that this sounds incredibly boring. Perhaps, but having at least part of your overall net worth in something like this kind of allocation model might not be a bad idea, in the long run.

Whether you are interested in implementing such a portfolio or not, does not really matter. It makes for a good exercise in constructing Python based

backtests. The 70/30 asset mix suggested by Buffett is just one example, and in our first model here we will use a slightly broader allocation.

Important to understand is that the point here is to demonstrate concepts, not exact rules. The approach used here is absolutely valid, but which ETFs you would chose and at which weights is more a matter of preference than anything.

The rules of this first ETF model are the following. We will use five ETFs to allocate our assets to. Each ETF will have a target weight, and at the beginning of each month we will reset the allocation to this target weight.

The model will hold the S&P 500 Index tracker, SPY, at a weight of 25%. On the bond side, we will have the long term 20 Year Treasure ETF, TLT, at a weight of 30% as well as the medium term 7-10 Year Treasury ETF, IEF, at a weight of 30%. We will also add some commodities, with the gold tracker GLD at 7.5% and the general commodity tracker DBC, also at 7.5%.

Now this should not be too hard to model. If you feel confident enough with the earlier lessons on backtesting, go ahead and build this code by yourself. It's really quite simple.

The actual code needed for this model is very brief. It does after all have very simple rules. The logic of what we are doing is as follows. In our initialization, we define a dictionary of ETF tickers and target weights. This is essentially our intended portfolio allocation. Then we schedule a monthly rebalance.

In the monthly rebalance, we loop through the dictionary, and set the target weight accordingly. And that's pretty much it.

First, I will show you the code for what was just described, along with the now standard piece of code at the bottom to fire off the backtest. The code below performs the entire backtest, but does not output any sort of results.

The results, as you can see when the backtest is fired off in the code, will

be stored in variable I called result. To demonstrate an often convenient way of working with backtests, I suggest you put this code in a Jupyter Notebook cell by itself, and execute it. Then we can analyze the results in the next cell later.

You may notice in the code here, that I used a bundle called ac_equities_db. This is a custom bundle, constructed as explained in chapters 23 and 24. We are reaching the limit to what can be done with free data sources like Quandl, and that's why I'm moving to custom bundles.

That means that you may need to find your own data source to replicate code from hereon in the book. I cannot provide that data for you, as that would breech all kinds of license agreements, but the later chapters in the book will explain how to connect your own data.

```
%matplotlib inline
import zipline
from zipline.api import order_target_percent, symbol,
schedule_function, date_rules, time_rules
from datetime import datetime
import pytz
```

```
from matplotlib import pyplot as plt
import pandas as pd

def initialize(context):
    # Securities and target weights
    context.securities = {
        'SPY': 0.25,
        'TLT': 0.3,
        'IEF': 0.3,
        'GLD': 0.075,
        'DBC': 0.075
    }
```

```
    # Schedule rebalance for once a month
    schedule_function(rebalance,
date_rules.month_start(), time_rules.market_open())
```

```
def rebalance(context, data):
    # Loop through the securities
    for sec, weight in context.securities.items():
        sym = symbol(sec)
        # Check if we can trade
        if data.can_trade(sym):
            # Reset the weight
            order_target_percent(sym, weight)
```

```
# Set start and end
start = datetime(1997, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2018, 12, 31, 8, 15, 12, 0, pytz.UTC)

# Fire off backtest
result = zipline.run_algorithm(
    start=start, # Set start
```

```
    end=end,  # Set end
    initialize=initialize, # Define startup function
    capital_base=100000, # Set initial capital
    data_frequency = 'daily',  # Set data frequency
    bundle='ac_equities_db' ) # Select bundle

print("Ready to analyze result.")
```

As you see, this code is very straight forward. Constructing models like this is quick, easy and painless. I also used a new concept in there, which I hope you already picked up from glancing the code.

Previously in chapter 5 we worked with lists, which are as the name implies just lists of any kind of stuff. In this case, I instead used a dictionary, something you may remember from the same chapter. As a brief reminder, lists are created by putting things in brackets, much like below.

```
some_list  =  [‘some text’, ‘some other text’, 5, 8.217]
```

A list can contain any type of variable or value, and above you see a mix of text and numbers, just to demonstrate this.

A dictionary on the other hand, can hold pairs of information, to be used as a

lookup table, and it's enclosed in curly brackets.

```
a_dictionary = {'one': 1, 'two': 2, 'three': 3}
```

A dictionary matches two items. A key and a value. As you see in the code for the ETF model above, we use strings as keys and weights as values in context.securitie s . Then later in the rebalanc e routine, we iterate this object using a very simple syntax.

```
for sec, weight in context.securites.items():
```

This way we can in a very fast and easy way go over each security in the dictionary, check the target weight and make the necessary trades.

But now are you probably wondering if this model is worth trading or not. When you see the actual results, you may very well be disappointed. This is not a model that attempts to grind out a maximum of return. This is a variant of what Mr. B suggested as a retirement saving scheme. Low returns, low risks.

Since you have already learned about how to visualize output from

backtests in chapter 8, you can apply the same logic to get a feel for how this simple strategy performed. As you will notice if you run the code segment above, after the backtest run it simply outputs a line of text to inform you that it's ready for analysis. Now you can make a new cell in the Notebook, below the backtest, and write something to visualize the output. Revert to chapter 8 if you need a refresher.

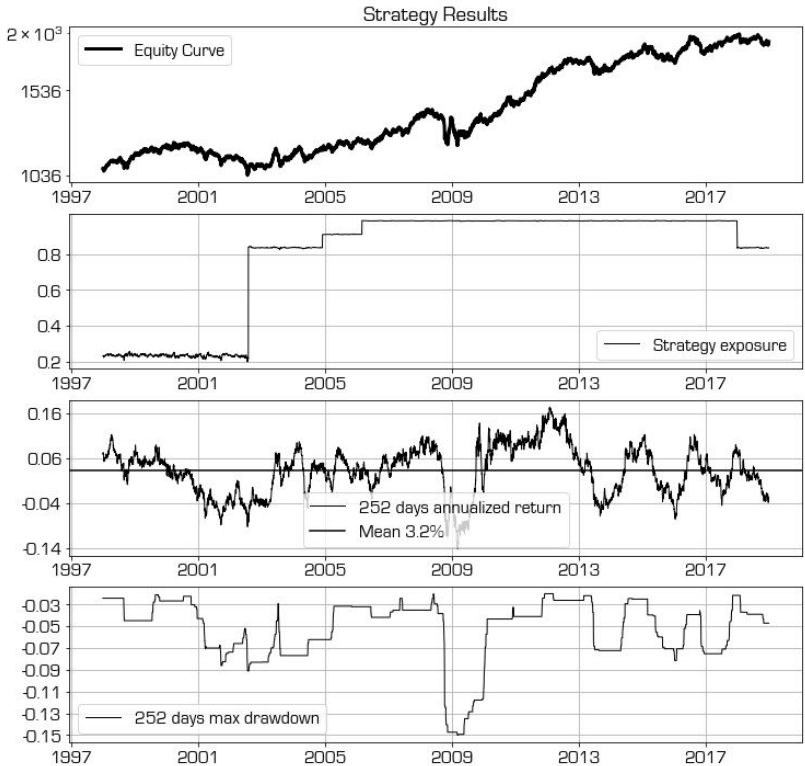


Figure 10-1 ETF Asset Allocation

So how do we interpret this? Clearly a graph like this does not hold sufficient information for any in-depth study, but we can still make out a few interesting observations. The first segment, the equity curve, seems to show a reasonably smooth and well behaving profile. Without looking too closely at the scale, it seems at least worthwhile to study further.

The second graph should raise an eyebrow. At least I included it in hopes that you will react to it. Did we not just make portfolio allocations that always add up to 100 percent? Why is the exposure level going up in steps?

This is an issue with ETFs which can cause some headache. Most ETFs haven't really been around that long. I still remember back in the later part of the 1990's when the first ETFs, such as the SPY and the QQQ were first launched. Only a handful have such long history. Most ETFs were launched in the past ten to fifteen years. That makes it a little difficult to build proper long term backtests in the ETF space.

So the reason why you see the exposure going up in steps is that from the start of the backtest, the only available ETF was the SPY. Then the bond ETFs were launched. Then the gold tracker, and finally the general commodity tracker.

The sparse availability of long term ETF time-series is something we need to take into account when constructing ETF

models. There is simply not enough data. The obvious danger is to build models on just a few years' worth of data, which may not have statistical significance or predictive value.

A common beginner mistake is to simply look at the end results of a model, without digging into the details of how you got there. The issue here, that we only have a very limited amount of historical data for some ETFs is but one of many that can arise from not paying attention to the details.

The third segment shows the annualized return, as we discussed earlier, this time showing a full year rolling window. This is just a simple visualization to show what kind of returns you may be looking at over time. While this sometimes spikes up even to the 15% range, we are seeing an average value of a little over 3%. Well, I did tell you that it was a slow moving model.

Yes, it's not a terribly exciting model, but that's not the purpose. It is after all an asset allocation model. But

making these kind of models serves a purpose. For one thing, most people should have some sort of longer term, stable investment portfolio which is not subjected to the kind of risks that shorter term trading is prone to. You could also use allocation models like this as a realistic alternative to trading, to be used as benchmark indexes and compared to your trading models.

11 Equities

In the brilliant book Liars' Poker (Lewis, 1989), Michael Lewis writes about his time as a trainee at Salomon Brothers and how no one wanted to work in equities. The reason stated was that if you work in equities, your mother will understand what you do for a living.

Equities seems on the surface as the easiest asset class. Something that everyone understands and can participate in. At first glance, this may very well be true. But make no mistake. It's in no way easier to make money in equities than in

other asset classes. And it is not easier to construct trading models for equities, at least not if you plan on doing it right.

As with most asset classes, equities have a few unique problematic aspects that you need to deal with. Since you made it this far into the book, it's a fair assumption that you actually plan to take quant modeling seriously and that means that we need to have a talk about equity methodology.

The Most Difficult Asset Class

Stocks are a curious asset class in that they are both the easiest asset class to understand and the hardest to trade.

We all know what stocks are. It's not exactly like the CDS market which everyone has spent years pretending to understand after they almost blew up the world. No, stocks are simple and straight forward. A business builds something, sells it and makes money. And we are

buying a part of that.

When we buy shares in a company, we are proud co-owners of the firm. But unless your name is Buffett, the percentage of your ownership share is likely so small that it's barely even theoretical. That part is not all that important though. What is important is that we know what we are buying, or at least we think we do.

You like your iPhone and you get your three balanced meals a day from McDonalds like most people. So you go and buy shares in Apple and Mickey Dee. You know and like their products and understand what they do.

This is the part that's easy. To gain a high level understanding of what the company does and what it means to buy shares of it. Unfortunately, this is just an illusion.

The fact that you like a company's products does not in any way have an impact on the future share price, as so many enthusiastic buyers of Private Media Group found out the hard way.

Unless you are part of the top management or on the board of directors, your knowledge of the firm isn't unique and does not give you an edge. And if you happen to be in those categories, there are some pesky little laws against trading on it anyhow.

It's not that it's impossible to figure things out about a company and trade based on your views of their business. Not at all impossible. But it takes a whole lot more work than tasting the latte before investing in Starbucks. If you are just a casual observer, even a loyal customer or a long term employee, whatever you think you know about the company is known by everyone else as well and won't help you. It will more likely do just the opposite.

This illusion of special knowledge is just one of the dangers of the stock markets. The other danger is much more severe. The problem is that stocks are stocks and they tend to move more or less the same way.

When the stock markets are in a bullish state, almost all stocks move up.

They also tend to move up on the same days, and down on the same days. Then a bear market comes along and they all move down at the same time.

Some stocks do better than others, but stock investing is generally a relative game. What that means is that you are really competing with the equity index. Obviously we all like to make money from an absolute point of view, but equity strategies tend to be at the mercy of the overall health of the markets. You can't realistically expect to see the same returns in a bull market and in a bear market, or to see the same type of strategies working in all market conditions.

The fact that stocks tend to move up and down at the same time means that they have a very high internal correlation. That means that, as a group, stocks are quite uniform. The real issue here is that we have a very limited diversification effect.

If you buy one stock, you have no diversification. If you buy a basket of 50 stocks, you have some diversification, but

you are still just holding stocks. Your main risk factor is how the overall stock market index is doing. This is the main problem with stocks, and why it's the most difficult asset class. Some stocks are better than others, but in the end they are all just stocks.

This is not to say that it's a bad asset class to trade. It's just different and you need to be aware of the relative nature of it.

A word on Methodology

Methodology matters. It's a really horribly boring word and it sounds like something that the mid office should take care of for you, but unless you work for a large bank that's not likely to be the case. If you perform your research with flawed methodology, you will get flawed results. So we will take a moment first to look at how the simulations in the equity section of this book are done before showing the

results. No, don't skip ahead. This is important.

There are two primary pitfalls when it comes to constructing equity models. The first has to do with deciding which stocks are possible to trade, and the second concerns dividends. You will need to properly handle both of these issues, or your backtesting will be all for nothing.

Equity Investment Universe

For equity portfolio type of models, you have a vast number of markets to choose from, in theory. To increase realism, you need to ensure that your algorithm only considers trading stocks which would reasonably have been considered in reality at the time.

It's easy to pick stocks that you know and are familiar with, say like Google, Apple and the like. The obvious problem here is that you know these stocks because they have strongly

increased in value. You probably didn't pick Global Crossing, Enron or Lehmann Brothers for your backtest, did you?

This is a very common mistake. Most people would instinctively pick a basket of stocks that they are familiar with and then make the logical leap of thinking that these are the same stocks you would have traded ten years ago. That's not terribly likely.

One solution to this would be to pick an index and trade the stocks in that index. But there is a trap here as well. If you were to pick the S&P 500 for instance, and then trade the constituent stocks, that might alleviate the issue somewhat. It would be logical to assume that a major index like this would have been a plausible pick even ten or twenty years ago. But only if you take the index joiners and leavers into account.

The current S&P 500 index constituents are not the same as the index's members ten years ago. Consider for a moment how an index, any index, is designed. Why the stocks are in there in

the first place.

For almost all indexes, the stocks became members primarily because they fulfilled certain market capitalization criteria. That is, they were suddenly worth enough to be considered. And how do you think that happened?

Stocks are included in an index because they have had strong performance in the past. It's hard to imagine these days, but Apple was once a tiny garage company run by a couple of hippies who made innovative products, did their own design and paid taxes. At one point their share price had moved up enough for them to get into small cap indexes. After a while longer of strong share price performance, they made their way into the mid-caps.

Much later, after they have already become one of the largest companies in the world, everyone dreams of having bought them 30 years earlier. It would be dangerous to allow a simulation such wishful thinking.

What we do here is instead to pick

an index, and then consider only stocks that were members of that index on any given day. The simulations here are aware of when stocks joined and left the index, and are only allowed to hold stocks that were really part of the index on the relevant day.

This aims to reduce the effect of so called survivorship bias. It can be a little tricky to implement, and that's probably the most difficult aspect of developing trading models for stocks. While this can be approached in multiple ways, you will find one possible solution in chapter 24, which deals with data and databases.

You could of course work with other types of data to achieve more or less the same thing. One way would be to look at historical market capitalization or traded volume to construct an investment universe which would be a reasonable approximation of what you may have considered at a given point in time.

Dividends

Next we need to deal with dividends. There are other things to adjust for as well, such as splits and similar corporate actions, but you generally don't really need to worry about those. Everything but cash dividends is easy and it's very likely that any stock price history you will ever look at is already adjusted for the rest. When a stock has a 2:1 split, you won't see the share price take a 50% nose dive. Instead, you see the history back in time automatically adjusted for you, so that the split does not have an impact. Splits and similar are simple events with simple solutions and they don't impact your profit or loss at all.

What you do need to worry about is cash dividends. There is no easy answer here, but there are various ways to deal with the issue. Clearly, the worst way to deal with it is to pretend that they don't exist. Over multiple years, dividends will have a significant impact.

There are two different methods for dealing with dividends. The easiest would be to use total return series. The total return series for a stock shows you, as the

name aptly implies, the total return you would have realized in the stock, assuming all dividends are directly reinvested into the same stock. So you are holding 200 shares of ACME Inc. trading at 10 dollars a share. Now there is a dividend of 50 cents, and you are the lucky recipient of a crisp Ben Franklin. The total return series assume that Mr. Franklin is immediately used to shop for more ACME at market.

To reflect this in the time series, the entire series back in time will now be adjusted so that the percent gain or loss on buying and holding the stock matches. Note that this means that if you check the price of the stock a year ago, or ten years ago, it may greatly deviate from the actual nominal price at the time. The time series now reflects price and dividend, assuming full reinvestment.

The other way might be easier to relate to, and is even more realistic, but can be a little tricky to model. This method assumes that when dividends are paid, they are actually paid in cash. Just like in real life. We leave the price series

alone and just credit the cash holdings with the appropriate amount.

Since we don't change the price series, the charts will reflect the actual prices in the markets, and not the artificially adjusted price that the total return method would present. We are also presented with the very real problem of what to actually do with new cash. They can drop in on the account when you least want them, and now you need to decide how to deal with it. Realistic, but potentially complex to model if your software is not already set up for it, and if you lack proper dividend data.

The total return method is good enough for the most part and very easy to model. The downside is that total return series are hard to come by for free. The usual suspects of free stock history providers on the internet are not likely to give it to you.

The second method is better and worth doing if you want to have a more realistic backtest. The good news is that Zipline, the backtester that I'm using for

the purpose of this book, can handle the dividends for us. We just need to make sure that the dividend data is supplied, and the logic is then done by the backtester.

You will find a detailed technical explanation of how to import your stock pricing data, as well as dividend data, in chapters 23 and 24.

12 Systematic Momentum

Momentum is a market phenomenon that has been working well for decades. It has been confirmed both by academics and practitioners and is universally known as a valid approach to the financial markets. It's also based on a very simple principle.

Slightly simplified, momentum is the principle that stocks that moved up strongly in the recent past are a little more likely to do better than other stocks

in the near future. Yes, we simply buy stocks that have had a good run. The trick is in finding a method of identifying the stocks to buy, and how to construct a portfolio from them and knowing when not to buy.

In this chapter we will construct a momentum model, step by step. While the concept itself is very simple, there are some details that can be quite important in arriving at a model which will be able to show stable long term performance.

The aim here is to show a solid approach that you can build upon, modify and improve as you wish. We are not talking about some sort of magical trading system to beat all trading systems. You will have to look for that in other trading books. Here we will deal with real life quantitative trading models. The purpose of this book is to teach methods and approaches, not to peddle some supposedly magical trading systems.

Replicating this

Model

Up until this point in the book, it has been reasonably easy to replicate the models and run the code by yourself. By easy, I mean that it didn't require any special data or unorthodox technical solutions.

Dealing with equities, does however require some complex solutions. There is one complication in this chapter which could potentially cause a bit of a headache, and that is universe selection. How we, from a technical point of view, solve the issue of which exact stocks we are able to trade.

The aim here, with this model and this chapter, is to limit the stock selection to the historical membership of the S&P 500 Index. We want to make sure that the model is aware, for each day, which exact stocks were members of that index, and only trade those.

This of course requires you to have such information available, and to find a solution for how to store it and how to

access it. My own preferred approach is to store this kind of data in a local securities database, something which is explained in more detail in chapter 24. Not all readers are likely to go the full nine yards and getting your own database, so I will offer an easier solution, along with that probably misused American sports reference which I really don't understand myself.

The solution I will show you here is based on having a local comma separated file with an updated index composition for each day where it has changed. The code, as you will soon see, will then fetch the index membership list for the closest prior date to the current trading day.

This added complication of universe selection, of deciding which stocks were possible or probable to be selected for trading in the past, is unique for equities. That is one of the main reasons why equity models are usually more complex from a technical point of view.

This may sound surprising, but for most readers of this book, it will be easier

to replicate the futures models later in this book, as the survivorship bias does not play in there the same way.

Momentum Model Rules Summary

- Trading is only done monthly.
- Only stocks in the S&P 500 will be considered.
- Momentum slope will be calculated using 125 days.
- Top 30 stocks will be selected.
- Weights will be calculated for these 30 stocks according to inverse volatility.
- Volatility is calculated using 20 day standard deviation.
- Trend filter calculated based on a 200 day average of the S&P 500 index.
- If the trend filter is positive, we are allowed to buy.
- Minimum required momentum value is set to 40.
- For each of the 30 selected stocks, if the stock has a momentum value higher than 40, we buy it. If not, we leave that calculated weight for that stock in cash.

- We sell stocks if they fall below the minimum required momentum value, or if they leave the index.
- Each month we rebalance and repeat.

Investment Universe

First we need to settle on an investment universe. That is, decide which stocks are eligible for selection. In my book *Stocks on the Move* (Clenow, Stocks on the Move, 2015), I used the constituent stocks of the S&P 500 index as investment universe. Not the current members of course, but the historical composition. On any given trade date, the model would check which stocks were part of the index on that day and consider only those stocks. This is to avoid, or at least reduce survivorship bias.

This way of limiting the investment universe is effective and fairly easy to implement. You could find other ways of accomplishing similar results, such as limiting based on market capitalization or trading volume, but if you have access to

historical index joiners and leavers information, this is a simple way to do it.

For the momentum models here, we will use the same methodology as in my previous book; limiting stocks based on historical index membership. As the S&P 500 is probably the most widely followed market index in the world, I'm going to stick to that index in this demonstration. But that's not to say that this index somehow produces better results. If you could really use the word better in such a context. It all depends on what you want to accomplish. What kind of return profile to you need. There is no 'better'.

As long as you have a broad enough index, with at least a few hundred available stocks to choose from, the same principles and logic should be applicable. Before asking whether or not the same model works on your own local market index, go ahead and try it out. Replicate what this book does, get the data, run the backtests.

I want you to learn ideas from this book. Then I want you to go out and test

variations of those ideas. See what you like and what you don't. I'm giving you the tools for learning, not exact rules to follow.

Momentum Ranking

Next we need a way to identify stocks to buy. There are many types of momentum analytics and most will show a fairly similar ranking. What we want to do here is to make a list of well performing stocks. How hard can that be?

The simplest way of doing this is to measure percent return over a given time frame. So we could take our 500 stocks, measure the difference between yesterday's price and the price of say, half a year ago, and rank them accordingly. After that, all we have to do is to start buying from the top of the list.

While such a simple concept could show decent returns, we are going to do things a little more complex here. The problem with the simple percent return is it does not take account just how the

returns happened.

If we simply rank stocks based on the percent return, we will likely end up with an over representation of really wild situations. We would see extremely volatile stocks which could gain or lose large amounts in short periods of time, or we could get takeover situations where the stock price suddenly made huge jumps. This is not really what momentum should be about.

While there are several valid ways of calculating stock momentum, I will use my own momentum score. My concept is simply a variation of well-established momentum calculations, but it's one that I believe holds some value. What I do is to measure the momentum using exponential regression slope, and then multiply it by the coefficient of determination. No, that's not as complicated as it sounds.

```
momentum_score = Annualized Exponential
Regression Slope * R2
```

Let's take that one step at a time. The difference between linear regression, which most are more familiar with, and

exponential regression is that while the linear regression slope is expressed in dollars, in the case of USD denominated financial markets, the exponential slope is expressed in percent. Linear regression wouldn't be very useful, unless of course every single stock has the same price to begin with, which is not overly likely.

Regression is a method of fitting a line to a series of observations. It aims at finding a line that fits the best. That does not mean that it necessarily fits perfectly, but we will get to that. What is important to understand is that we are trying to fit a line to the data, in our case the daily stock prices. To draw a line, we need a slope and an intercept. But we are not really interesting in actually drawing the line, we just need to know the slope. That is, the trend.

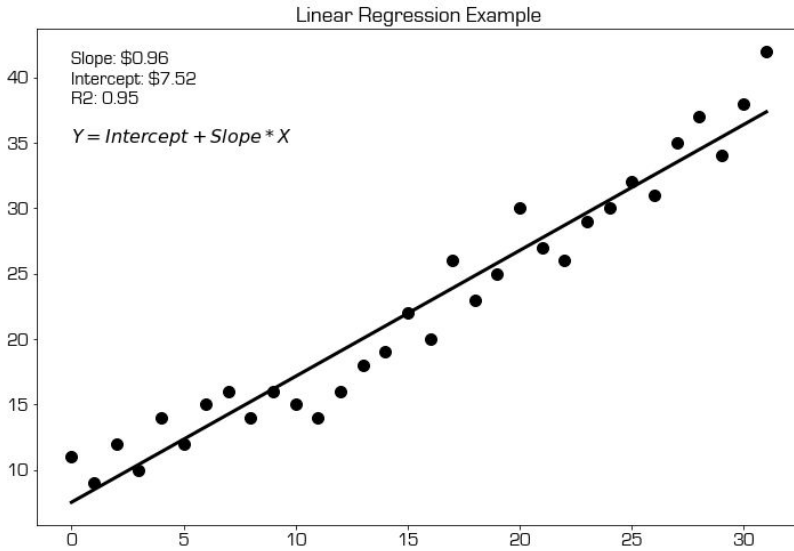


Figure 12-1 Linear Regression

A linear regression slope tells us how many units, in the case of American stocks that means dollars, the regression line slopes up or down per day. So if the same trend continues, the slope would tell us how many dollars per day we should expect to gain or lose on average.

But such a measurement is not very useful for our purposes. A stock with a current price of 500 will show very different numbers than a stock priced at 10. If both are showing a current linear regression slope of 1 dollar, that means two very different things. It means that the stock priced at 500 is moving very

slowly, while the stock priced at 10 is moving very fast. Regression is useful, but not necessarily linear regression.

If we instead look at exponential regression, things get more interesting. The exponential regression slope tells us how many percent up or down that the line is sloping. That means that if you draw an exponential regression slope on a regular, linear scale graph, you would get a parabolic curve.

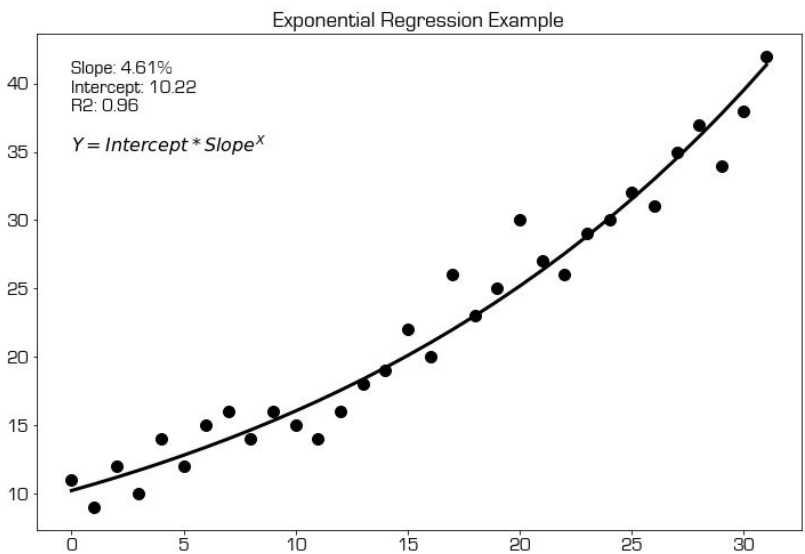


Figure 12-2 Exponential Regression

What makes exponential regression so much more useful is that we can now compare it across multiple stocks,

regardless of their base price. The exponential regression slope can be a little hard to relate to, as it's usually a very small decimal number. After all, how many percent per day do you really expect a stock to move?

For instance, you might end up calculating an exponential regression slope of 0.0007394. Tough to remember. But it's much easier to relate to if you annualize it. That is, calculate what this daily slope means on a yearly basis, should the current trend persist for that long. Keep in mind that we don't actually expect that to happen. That's not important. We annualize the number to make it easier to relate to.

So take that number again, 0.0007394, and annualize it. The math for that is easy.

$$\left((1 + 0.0007394)^{252}\right) - 1 = 0.2047$$

That means that now we have a stock which has a positive trend equivalent of about 20.5% per year. That should be a bit easier to relate to.

The reason we raised that number to the power of 252 is that we assume 252 trading days in a year. If we move up by 0.07394% per day for 252 days in a row, we will end up gaining about 20.5%. I told you that the math would be easy.

But just using the exponential regression slope alone does not tell us anything about how well this line fits. An extreme example would be where the price went absolutely sideways, and then made a 50% single day jump on a takeover announcement.

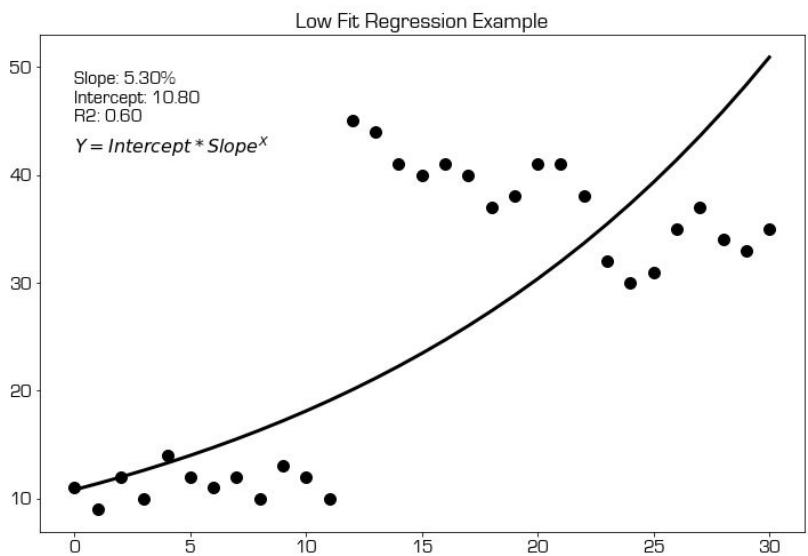


Figure 12-3 Low Regression Fit

The exponential regression value

will likely be very high. But that's not the kind of situation that we are looking for. The same, but slightly less extreme, goes for very volatile stocks. The line may not fit very well there either. We need to compensate for this.

A very important concept in finance is that volatility is, for lack of a better word, bad. It's not possible to avoid volatility, but you need to be compensated for it. All returns need to be viewed in the light of the volatility it took to achieve it. Therefore, we need to adjust our exponential regression slope for this. While there are many ways of looking at volatility, in this particular case, the coefficient of determination is a great tool.

This analytic, commonly denominated R^2, tells us how well our regression line fit the data. If it's a perfect fit, with all observations exactly on the line, the R^2 will be exactly 1. If it's utterly random noise, and the line does not fit in any way whatsoever, the R^2 value will be 0. Therefore this analytic will always be between 0 and 1. The better the fit, the

higher the value.

What we will do is simply to take the annualized regression slope, and multiply it by the R^2. Effectively, we are punishing stocks that are more volatile. Stocks that are moving up in a smooth trend will have a higher R^2 value and their slope won't take too much of a hit. But in the example of the takeover situation mentioned earlier, the R^2 will be very low and greatly reduce the slope value.

So how do you calculate exponential regression? If you want the big scary looking formula, you will have to look it up yourself. If you are the kind of guy, or gal, who is interested in a complex looking formula, you will either already know it or at least you will have no problem finding it. It's not exactly secret information. Instead, I will show you how to use Python to calculate the result.

```
from scipy import stats
def momentum_score(ts):
    """
    Input:  Price time series.
```

```
    Output: Annualized exponential regression slope,
          multiplied by the R2
    """
    # Make a list of consecutive numbers
    x = np.arange(len(ts))
    # Get logs
    log_ts = np.log(ts)
    # Calculate regression values
    slope, intercept, r_value, p_value, std_err =
stats.linregress(x, log_ts)
    # Annualize percent
    annualized_slope = (np.power(np.exp(slope), 252) -
1) * 100
    #Adjust for fitness
    score = annualized_slope * (r_value ** 2)
    return score
```

Here is what the function above does. The expected input here, the variable `ts`, is a time series. Throw your time series at this function, and you get the momentum value back.

Step by step. First we calculate a natural log series from the price time series we gave it. That's because the easiest way to calculate exponential regression is to just do a regular linear regression based on the log values.

```
log_ts = np.log(ts)
```

To calculate regression, we need

two axes. One is of course the price observations. The other is just a constantly incrementing series. 1, 2, 3, 4 and so on. After all, the price observations are at equal distances. We have one observation per day. Using date values or any other numbers makes no differences, so we will use a neat **Numpy** function called arang e . No, that's not misspelled.

This function, arang e , simply returns a series of the same length as the input, with the values 0, 1, 2, 3, 4 … up until the full length. So we very easily get a matching x-axis for our regression calculation this way. By default, arang e will give you a zero based list, though you could provide a start and stop if you like. In this case, it really does not matter.

```
x = np.arange(len(ts))
```

Now we have two series. One with log values of the prices and the other with an equal spaced x-series. That means that we can now do the regression math. But of course, we don't have to reinvent the wheel here. No need to write the standard regression formula ourselves. Just use the

one built into the statistics library **scipy** , which is short for Scientific Python. Remember to import it, at the top of the code.

```
from scipy import stats
```

Using the pre-built function in this library, stats.linregres s , we can output the values associated with regression. As you see, in one line we get five values; **slope** , **intercept** , **r-value** , **p-value** and **standard error** . All we need here is the **slope** and the **r-value** .

```
slope , intercept , r_value , p_value , std_err = stats .
linregress ( x , log_ts )
```

The following row is important to understand. This is where we transform the slope into something we can use, and relate to. Breaking it down to its parts we get this. Start with np.exp(slope) . That will give us the percentage slope of the exponential regression line, telling us how many percent up or down per day it moves. Then we raise it to the power of 252, arriving at an annualized number. Multiply by 100 to get a number that's easier to relate to.

Ok, so some of you are now asking yourselves the obvious question. Why we would need to be able to relate to the number, when computer code does the interpretation and trading. Well, during development and testing, you will likely want to output this number, or graph it. And if this is transformed into something that makes intuitive sense, such as a yearly number, you will much easier be able to spot issues and identify opportunities. But sure, it makes no difference for the algo per se.

```
ann_slope = ( np . power ( np . exp ( slope ), 252 ) - 1 ) * 100
```

And then, finally, we have only one more thing to do. Multiply it with the R2, before returning the result. We got the **r-value** before, so just square it, multiply the annualized slope with it, and return the result. As you see in the code just below, we can square a number using the double multiplier syntax.

```
return annualized_slope * ( r_value ** 2 )
```

And that's how we calculate the momentum score used for this model.

Position Allocation

The most common way of allocating to positions is to base it on volatility. Perhaps I should qualify that statement. Volatility based allocation is most common in the professional, quantitative side of the financial industry. Among hobby traders and traditional asset managers, equal sizing remains most prevalent.

If we want to take on an approximate equal risk in a number of positions, we would normally look at how volatile they are and allocate accordingly. A more volatile stock gets a smaller weight. Slower moving stocks get a higher weight. The rationale behind this is simple. As we don't have any reason for assuming that any of the selected stocks will perform better or worse than any other, we want to give them an equal chance.

If we put the same weight on all stocks, that is allocate the same amount of money to each, we would get a portfolio

geared towards the most volatile ones. If some stocks tend to move 4-5 percent per day, and others only half a percent on a normal day, the more volatile ones will drive the portfolio. No matter how well the slow moving stocks do, they just wouldn't have the same chance to make a dent in the bottom line.

The most important thing is that you understand the concept here. We are buying different amounts of each stock, but the purpose is to allocate approximately the same risk to each. The volatility serves here as a proxy for risk. The more volatile the stock, the less of it we buy.

Volatility can be measured in a few different ways. In my previous books, I kept things a bit simple by using Average True Range (ATR) as a volatility measurement. It's a concept that's well known in the retail trading space and it has been used for a long time. ATR is part of the old school technical analysis toolkit. Much of this toolkit is highly simplistic, as the math was done with pen and paper in the old days. Lack of

computing power meant that everything had to be simple.

The ATR measurement is perfectly adequate for most retail traders, and there really is not anything wrong with it. Having said that, as this is a book leaning more towards the quant spectrum, the models here will use a different measurement, one that you will see more on the professional side of the industry.

The basis of our volatility calculations will be standard deviation. The main source of confusion when using standard deviation seems to come from the most basic question of what exactly we measure standard deviation of.

Clearly, it won't be helpful to measure standard deviation of prices themselves. This goes back to the same issue as we looked at earlier, in regards to regression. The base price of stocks can be very different, and we need to normalize for that somehow. The dollar variation of a stock trading at 500 and one trading at 10 can't be directly compared. But the percentage variation can.

We will therefore use daily percentage change as a basis for calculating the standard deviation, and use that as a proxy for volatility. In the financial industry, there are multiple ways of measuring volatility, but for our purposes here there is really no need to complicate matters. If you like, you could add a smoothing factor, such as exponentially weighted moving average of standard deviation, which is quite common. But a simple standard deviation measurement of percentage returns will do just fine for this model.

Luckily, calculating this is extremely simple in Python. We can do it in one single line of code. For the sake of convenience, we will make a function to use in the momentum trading model.

```
def volatility(ts):
    return
ts.pct_change().rolling(vola_window).std().iloc[-1]
```

This function takes a price time series as input. That one line of code there shows the beauty of Pandas. This line of code does a few things for us. First, it calculates the percent difference between

consecutive days. Then it gets a rolling window of that time series, the length of which is set by the model input factor vola_windo w , as you will see at the top of the complete source for the model later in this chapter.

We then calculate the standard deviation of this rolling window, and finally get the last data point of that calculation and return it. All in one single line. Remember that we can slice off the last data point using the syntax [-1] .

Momentum Model Logic

Now that you understand the momentum ranking logic and the volatility measurement that we need, it's time to construct a fully working momentum model. In my books, I always try to make clear that the models I present are meant as teaching tools. What I show are functioning models. The backtests are real. Well, valid anyhow. It would perhaps

be a bit far to refer to any backtest as ‘real’.

But I never tried to represent them as the Clenow Amazing Super System. Most of what I write is known by a lot of people in the investment management community. What I add is to try to make it more accessible, to teach a wider group of readers. I do try to add my own flair to things, but it’s important not to confuse my demonstration models with the various claims out there to have secret and near magical trading systems which makes everyone millions and millions. Which of course will be sold to you for a few thousand bucks.

My models are not revolutionary. They are not meant to be the best possible way of doing things. They are teaching tools, explaining methodology you need to understand in order to develop industry grade strategies. I want you to understand the ideas and concepts well enough to build your own models, and expand your toolbox as an algorithmic trading professional. All models presented in this book can be improved upon, and I very

much encourage you to attempt to do so.

The purpose of this particular model is to capture medium to long term momentum in the American stock markets. I want this model to be easy to manage, so we will only trade once a month. We are going to leave everything alone for an entire month at a time. This has a couple of advantages to higher frequency trading models. It allows a retail investor who has a day job to implement the rules without being overloaded with work. It also cuts down on turnover, which both reduces trading costs and more importantly reduces capital gains taxes, should you be unlucky enough to live in a jurisdiction that has such taxes.

Our stock selection will be based entirely on the momentum score described earlier, using a time window of 125 days for calculation.

What I have come to realize in the past when writing about model settings, is that some readers tend to see the demonstration settings as recommended,

or even the best possible settings. That's never the case. Most of the time when demonstrating models, I deliberately chose middle of the road kind of settings. I pick them more or less at random, from a set of reasonable values. The point is for you to try your own variations, and arrive at settings and tweaks which fits your own way of trading and your own requirements.

After all, the reason why I'm showing you all the source code in this book is so that you can test variations, take what you like and discard the rest.

Before I tell you the exact parameters we are going to use, I want to tell you why I have picked them. The logical thing to do would be to pick the best looking parameters, right? The ones that make the strategy look the best. Just run an optimization, fit the 'best' parameters and show the result. No, I'm not going to do that.

If I just showed you a curve fitted model, it wouldn't be all that useful. It would make things look great in a book,

but it may not have much bearing on reality, nor would it be much of a learning experience. Instead, I'm going to show you a version of this momentum model which quite deliberately will not use the 'best' parameters, if there even is such a thing. The model here is meant for you to improve upon yourself.

Now, for this demonstration, I will use a momentum score of 125 days, meant to roughly represent half a year.

For this model, we are going to have a set target number of stocks. We need a reasonable diversification and for that more than just a handful of stocks are required. So we will set the target number of stocks to 30. The number is chosen because it's reasonable. If we only hold 10 stocks, the single stock event risk would be quite high. A freak event in a single stock would have a very large impact on the overall results. That risk is mitigated when holding a broader portfolio. If we would pick too many stocks, we would have more work in executing and monitoring the portfolio and there would be a risk that the quality would suffer due

to the inclusion of so many stocks.

All trading is done monthly, to keep costs and taxes reasonable. At the start of the backtest, we are going to rank the stocks in our universe based on the momentum score, and buy the top 30 stocks in the list, provided that they have a momentum score higher than 40.

This fairly arbitrary number, is to ensure that we are not buying flat or negative stocks. The idea is to buy strong stocks, not stocks that are falling less than others. If there are not enough positive momentums stocks at the moment, there is no point in buying the least bad ones.

The shorter momentum period you use, the more extreme the momentum score will be. That's because short term events are then extrapolated on a yearly basis, and can result in extreme numbers. So a filter based on the absolute level of momentum like this needs to be set higher for shorter momentum windows. Good to know if you want to experiment with the code yourself with setting different minimum required momentum levels.

While we initially buy the top ranked 30 stocks, we are not replacing all stocks with the current top list each month. We could do that, but it would likely result in a bit unnecessary trading. Consider a stock which keeps moving between position 30 and 31 on the ranking list. Would it really make sense to keep buying and selling due to such small ranking changes?

What we will do instead, is to keep our positions as long as they still have a momentum score of above 40 when it's time for the monthly rebalancing. We will also sell stocks that left the index, which helps keep the backtest a bit more realistic.

Downside Protection

A well-known phenomenon is that momentum strategies tend to suffer in bear markets. Not just lose money, but often suffer substantially worse than the overall market. That's why many momentum ETFs you might see are

looking a little weak over the long run. They take an outsized beating during bear markets and struggle to make up for it during bull runs.

One way of dealing with this is to use an index level trend filter. In a similar model for my book Stocks on the Move, I used simple 200 day moving average, or rolling mean to use the fancy pants data sciency name for it. The idea is to simply not allow any new buys when the index is below the 200 day rolling mean of the daily closing price. That does not mean that current holdings are sold because the index dipped into bear mode, just that new stocks are not bought.

There is valid criticism of such an approach though. Some may argue that using such a crude trend filter is in itself a severe curve fitting. The gist of the argument is that we already know from experience that using such a long term trend filter will greatly mitigate damage from the two major bear markets of our generation. The question is of course if that has any predictive value in terms of avoiding the next.

The other point one could make is that if you include such a trend filter, you are really looking at two strategies, and not just one. There is a momentum strategy and a market timing strategy. Whether or not you want to use a trend filter is really a personal preference in the end.

For the model in this chapter, I won't employ such a trend filter but I will post a version of on the book website, **www.followingthetrend.com/trading-evolved** , where you can see how that could easily be done. This also helps keep the code amount down in this chapter, and should make it easier to absorb the information.

If you try it out and enable this trend filter, you will find that the performance improves. But again, you would need to decide for yourself if that amounts to cheating and curve fitting or not.

We do have some downside protection for this model though. We are relying on the minimum required

momentum score, as explained earlier, to scale us out of the market in times of distress. As the general risk climate out there changes, and stocks start to fall, there will be fewer and fewer stocks available with sufficient momentum and we will automatically start scaling out at rebalance time. This strategy can theoretically be anywhere from 100% invested, to holding only cash.

In real life, holding large amounts of cash is almost always a bad idea. There was a time when you could be excused for viewing cash in a bank account as safe. But as someone who gets PTSD flashbacks from the number 2008, I can assure you that this is not the case anymore. People forget fast, unfortunately. There were a few weeks in 2008 where it seemed like any bank could go under at any time. When we were all hustling to move cash holdings, or overnight depos, from bank to bank at the end of business every day. Trying to pick the banks that are most likely to survive one more day.

The strategy we are looking at here scales out of stocks during bear markets,

which of course tends to be when banks are more vulnerable than normal. Just imagine having made the right call and sold all your stocks early in 2008, holding cash and feeling smart, until you realize that all your cash was with Lehmann and now you are broke.

If your bank or broker goes poof, you will most likely recover any securities you held. Most likely. In case of fraud, anything can happen. But in the much more common case of greed mixed with incompetence, your securities will be returned to you when the lawyers are done. The cash however, at least any amount exceeding government deposit guarantees, has gone on to the happy hunting grounds.

For that reason, you may want to look into ways to avoid or at least reduce holding cash. When there otherwise would be excess cash available on the books, one way would be to buy a fixed income ETF. For instance, putting excess capital in the 7-10 year treasury ETF (IEF). Where you want park your cash depends on what kind of risk you want to

take. If the purpose is to take minimal risk, pick something short end like the 1-3 year treasury ETF (SHY). If you are happy to take on some price risk in the bond market, you could go with a longer duration vehicle like the 20+ year treasury ETF (TLT).

I will keep it clean here though, and let the model stay in actual cash. I would encourage you to take the code here, modify it to add cash management and see how it impacts the results.

Momentum Model Source Code

When reading the source code section of this chapter, or for the later chapters as well for that matter, it may help if you are by a computer and can open the actual source files in front of you. You can of course do without that and just skip back and forth between looking at the text and the code segments of this chapter, but it may be easier by a

computer. If this is your first read through this book, you may want to simply skip this section for now, and return to it later.

In order to actually run this code, you would need to obtain your own data from a data provider and ingest the data to Zipline, which is described more in detail in chapter 23, as well as getting index membership data as we discussed earlier in this chapter.

In case you are wondering why I don't simply provide this data on my website, so that you can run the code right away, the reason is that I would rather avoid spending the next year or so fending off law suits.

You can download the full source code for this model, as well as for anything else in this book from **www.followingthetrend.com/trading-evolved** . But sadly not the price data.

The full source is at the end of this sector, but first have a look at interesting individual bits of it. At the top, as always, we import a bunch of libraries that we will have use for later on. After that, I've

put the model settings. Having them right up on top makes it easy to try model variations, changing things around without having to touch the actual code.

```
%matplotlib inline

import zipline
from zipline.api import order_target_percent, symbol,
\
    set_commission, set_slippage, schedule_function, \
    date_rules, time_rules
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import pyfolio as pf
import pandas as pd
import numpy as np
from scipy import stats
from zipline.finance.commission import PerDollar
from zipline.finance.slippage import
VolumeShareSlippage, FixedSlippage

"""
Model Settings
"""
intial_portfolio = 100000
momentum_window = 125
minimum_momentum = 40
portfolio_size = 30
vola_window = 20

"""
Commission and Slippage Settings
```

```
"""
enable_commission = True
commission_pct = 0.001
enable_slippage = True
slippage_volume_limit = 0.025
slippage_impact = 0.05
```

In the next section of the code, you will find a few helper functions. Tools really, do accomplish specific and limited tasks for us.

The first one is the momentum score calculations, and that has already been discussed earlier in this chapter.

```
def momentum_score(ts):
    """
    Input:  Price time series.
    Output: Annualized exponential regression slope,
          multiplied by the R2
    """
    # Make a list of consecutive numbers
    x = np.arange(len(ts))
    # Get logs
    log_ts = np.log(ts)
    # Calculate regression values
    slope, intercept, r_value, p_value, std_err =
stats.linregress(x, log_ts)
    # Annualize percent
    annualized_slope = (np.power(np.exp(slope), 252) -
1) * 100
    #Adjust for fitness
    score = annualized_slope * (r_value ** 2)
```

```
    return score
```

Second, the volatility calculation, which we will use for position sizing and which has also been explained in this chapter.

```
def volatility(ts):
    return
ts.pct_change().rolling(vola_window).std().iloc[-1]
```

And third, there is a helper function for all of us with short attention span. The next function will be called once a month, and it will simply output the percentage return for the previous month. We're not using that for anything, but it will give you something to look at while the backtest runs. Later in the book, we'll get to how you can make a bit fancier progress output.

```
def output_progress(context):
    """
    Output some performance numbers during backtest
run
    This code just prints out the past month's
performance
    so that we have something to look at while the
backtest runs.
    """
```

```
    # Get today's date
    today = zipline.api.get_datetime().date()
```

```
    # Calculate percent difference since last month
    perf_pct = (context.portfolio.portfolio_value /
context.last_month) - 1
```

```
    # Print performance, format as percent with two
decimals.
    print("{} - Last Month Result: {:.2%}".format(today,
perf_pct))
```

```
    # Remember today's portfolio value for next
month's calculation
    context.last_month =
context.portfolio.portfolio_value
```

Once we've gotten past these things, we're getting to the actual trading simulation. And the first part of that, just as before, is the startup routine. In `initialize`, we set the commission and slippage, if enabled, and then schedule a monthly rebalancing routine.

But we do one more important thing here. Do you remember in the beginning of this chapter how I mentioned that you need to care of historical index membership? Here, the code is assuming that you have a CSV file with this data. In

this startup routine, initializatio n , we read this file from disk, and store in the contex t object so that we can easily access it during the backtest.

```
"""
Initialization and trading logic
"""
def initialize(context):
```

```
    # Set commission and slippage.
    if enable_commission:
        comm_model = PerDollar(cost=commission_pct)
    else:
        comm_model = PerDollar(cost=0.0)
    set_commission(comm_model)
```

```
    if enable_slippage:

slippage_model=VolumeShareSlippage(volume_limit=slippage
price_impact=slippage_impact)
    else:
        slippage_model=FixedSlippage(spread=0.0)
    set_slippage(slippage_model)
```

```
    # Used only for progress output.
    context.last_month = intial_portfolio
```

```
    # Store index membership
    context.index_members = pd.read_csv('sp500.csv',
index_col=0, parse_dates=[0])
```

```
```

```
    #Schedule rebalance monthly.
    schedule_function(
        func=rebalance,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )
```

After this, we are getting down to business. Next comes the monthly rebalance routine, and that, as they say on Cribs, is where the magic happens.

Remember that we scheduled rebalanc e to run once at the start of every month. The first thing we'll do in this routine is to print out the performance since last month, using the helper function we saw just earlier in this section.

```
def rebalance(context, data):
    # Write some progress output during the backtest
    output_progress(context)
```

Next we need to figure out which stocks were in the index on this trading day, so that we know which ones to analyze for momentum. This could easily be done in one line of code, but I'll break it up into multiple rows, just to make it easier to follow.

First, get today's date. By today in this context, I mean the current date in the backtest of course.

```
# First, get today's date
today = zipline.api.get_datetime()
```

Remember that we have already fetched and stored all the historical index compositions in initializ e . We read it from disk and stored in a variable which we called context.index_member s , and we know that this variable contains a **DataFrame** with a date index for every date where the index changed, and a single column containing a comma separated text string with stock tickers.

Now we're going to first get all rows prior to the today's date.

```
# Second, get the index makeup for all days prior to today.
all_prior = context.index_members.loc[context.index_members.index < today]
```

Next we'll located the last row and the first column. The last row because that would be the latest date, and as we previously got rid of all dates higher than today, we now have the last known index

composition date for the backtest. The **DataFrame** only contains one column, and as always we can access the first column by referencing the number zero.

```
    # Now let's snag the first column of the last, i.e.
latest, entry.
    latest_day = all_prior.iloc[-1,0]
```

Now we have a long comma separated text string which we need to split into a list.

```
    # Split the text string with tickers into a list
    list_of_tickers = latest_day.split(',')
```

But it's not a list of stock tickers that we need, but a list of Zipline symbol objects. Easily done.

```
    # Finally, get the Zipline symbols for the tickers
    todays_universe = [symbol(ticker) for ticker in
list_of_tickers]
```

All those lines of code will step by step get us the relevant list of stock symbols which were part of the index, and by our model rules are eligible to be traded. As I mentioned earlier, I broke this logic up into many rows to make it easier to follow. But we could have just as well done the entire thing in one line of code, like this below.

```
    todays_universe = [
        symbol(ticker) for ticker in

context.index_members.loc[context.index_members.index
< today].iloc[-1,0].split(',')
    ]
```

Fetching historical data for all these stocks only takes a single row. There are 500 stocks in the index, but we can fetch all the historical close prices in this one row.

```
    # Get historical data
    hist = data.history(todays_universe, "close",
momentum_window, "1d")
```

Since we made a helper function earlier which calculates momentum score, we can now apply this function to the entire set of historical prices, again in a single row. The line below is very clever and a great part of how Python works. From the previous row, we have an object which contains historical pricing for about 500 stocks. The one line below will result in a sorted table of ranking score for all of them.

```
    # Make momentum ranking table
    ranking_table =
hist.apply(momentum_score).sort_values(ascending=False)
```

That means that we now have the analytics we need to figure out our portfolio. We have the momentum ranking table, and that's the core part of the logic.

With portfolio models, it's generally easier to first figure out what to exit, and then figure out what to enter. That way, we know how much we need to buy to replace closed positions. Therefore, we'll loop through all the open positions, if any, and see which to close and which to keep.

```
    """
    Sell Logic

    First we check if any existing position should be
sold.
    * Sell if stock is no longer part of index.
    * Sell if stock has too low momentum value.
    """
    kept_positions =
list(context.portfolio.positions.keys())
    for security in context.portfolio.positions:
        if (security not in todays_universe):
            order_target_percent(security, 0.0)
            kept_positions.remove(security)
        elif ranking_table[security] <
minimum_momentum:
            order_target_percent(security, 0.0)
            kept_positions.remove(security)
```

Now we have sold the positions that no longer qualified. Note how we first reacted a list of all the stocks in the portfolio, and then as any stock either dropped from the index or failed to maintain sufficient momentum, we closed it and removed it from the list of kept stocks.

After having closed positions we no longer want, it's time to decide which stocks should be added.

```
    """
    Stock Selection Logic
```

```
    Check how many stocks we are keeping from last
month.
    Fill from top of ranking list, until we reach the
    desired total number of portfolio holdings.
    """
    replacement_stocks = portfolio_size -
len(kept_positions)
    buy_list = ranking_table.loc[
        ~ranking_table.index.isin(kept_positions)]
[:replacement_stocks]
```

```
    new_portfolio = pd.concat(
        (buy_list,

ranking_table.loc[ranking_table.index.isin(kept_positions)])
    )
```

The stock selection logic above first checks how many stocks we need to find. That'd be the difference between the target portfolio size of 30, and the number of stocks we're keeping from the previous month.

One of those rows may require an additional explanation. This row below, broken into two rows to make it display better in this book, uses some clever Python tricks.

```
    buy_list = ranking_table.loc[
        ~ranking_table.index.isin(kept_positions)]
[:replacement_stocks]
```

The variable replacement_stocks, calculated just earlier, tells us how many stocks we need to buy, to replace those that were sold. The variable ranking_table holds a list of all stocks in the index, sorted by the momentum score. What we want to do is to pick stocks from the top of the ranking list, but without picking stocks which we already own and want to keep for the next month.

The first part of this row filters ranking_table to remove stocks which we

intend to keep, so that we don't select them again. We do that with the tilde sign, ~, which is the logical equivalent of not. That means that ranking_table.loc[~ranking_table.index.isin(kept_positions)] will give us the stocks in the sorted ranking table, but without those that are to be kept. From there, we simply slice it as before, using the object[start:stop:step] logic, starting from the beginning and stopping at the number of stocks we want. Easy, now that you know how, right?

Now we get the top ranked stocks in the momentum score list, excluding those that we already have, until we've filled up the target 30 stocks. Then combined the list of the buy list with the stocks that we already own, and presto, there's our new target portfolio.

But wait, we haven't traded yet. No, and we haven't even calculated trade sizes. Well, that's the only part left in this model.

We need to calculate the target weights for each stock, and place the order. Remember that we are using

inverse volatility to scale position sizes. Each month we are recalculating all weights, and rebalancing all positions. That means that we are adjusting size for existing stocks as well.

Step number one would be to make a volatility table for all the stocks, using the volatility helper function we saw just earlier. After inverting these numbers, we can calculate how large each position should be, as a fraction of inverse volatility to the sum of the inverse volatility for all selected stocks. And now we have a table with target weights.

```
    """
    Calculate inverse volatility for stocks,
    and make target position weights.
    """
    vola_table =
hist[new_portfolio.index].apply(volatility)
    inv_vola_table = 1 / vola_table
    sum_inv_vola = np.sum(inv_vola_table)
    vola_target_weights = inv_vola_table / sum_inv_vola
```

```
    for security, rank in new_portfolio.iteritems():
        weight = vola_target_weights[security]
        if security in kept_positions:
            order_target_percent(security, weight)
        else:
```

```
        if ranking_table[security] >
minimum_momentum:
            order_target_percent(security, weight)
```

Having made a table of target weights, we can now go over the selected stocks on by one, and make the necessary trades. For stocks we already own, we just rebalance the size. If the stock is a new entry into the portfolio, we first check if it has sufficient momentum. If it does, we buy it. If not, we skip it and leave the calculated weight in cash.

We calculated the target weights based on arriving at 100% allocation for all 30 stocks. But as can be seen here, we might not end up buying all 50, if some fail the momentum criterion. That's by design. The space in the portfolio reserved for a stock that does not make the momentum cut, is then simply left in cash.

This acts as a risk reduction mechanism during bear markets. If there are not enough well performing stocks to buy, this model will start scaling down exposure, and it can go all the way down to zero exposure during prolonged bear market periods.

Here is the complete source code for the entire model.

```
%matplotlib inline

import zipline
from zipline.api import order_target_percent, symbol,
\
    set_commission, set_slippage, schedule_function, \
    date_rules, time_rules
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import pyfolio as pf
import pandas as pd
import numpy as np
from scipy import stats
from zipline.finance.commission import PerDollar
from zipline.finance.slippage import
VolumeShareSlippage, FixedSlippage

"""
Model Settings
"""
intial_portfolio = 100000
momentum_window = 125
minimum_momentum = 40
portfolio_size = 30
vola_window = 20

"""
Commission and Slippage Settings
"""
enable_commission = True
```

```
commission_pct = 0.001
enable_slippage = True
slippage_volume_limit = 0.025
slippage_impact = 0.05

"""
Helper functions.
"""

def momentum_score(ts):
    """
    Input:  Price time series.
    Output: Annualized exponential regression slope,
          multiplied by the R2
    """
    # Make a list of consecutive numbers
    x = np.arange(len(ts))
    # Get logs
    log_ts = np.log(ts)
    # Calculate regression values
    slope, intercept, r_value, p_value, std_err =
stats.linregress(x, log_ts)
    # Annualize percent
    annualized_slope = (np.power(np.exp(slope), 252) -
1) * 100
    #Adjust for fitness
    score = annualized_slope * (r_value ** 2)
    return score

def volatility(ts):
    return
ts.pct_change().rolling(vola_window).std().iloc[-1]

def output_progress(context):
```

```
    """
    Output some performance numbers during backtest
run
    This code just prints out the past month's
performance
    so that we have something to look at while the
backtest runs.
    """

    # Get today's date
    today = zipline.api.get_datetime().date()

    # Calculate percent difference since last month
    perf_pct = (context.portfolio.portfolio_value /
context.last_month) - 1

    # Print performance, format as percent with two
decimals.
    print("{} - Last Month Result: {:.2%}".format(today,
perf_pct))

    # Remember today's portfolio value for next
month's calculation
    context.last_month =
context.portfolio.portfolio_value

"""
Initialization and trading logic
"""
def initialize(context):

    # Set commission and slippage.
```

```
    if enable_commission:
        comm_model = PerDollar(cost=commission_pct)
    else:
        comm_model = PerDollar(cost=0.0)
    set_commission(comm_model)
```

```
    if enable_slippage:

slippage_model=VolumeShareSlippage(volume_limit=slippage
price_impact=slippage_impact)
    else:
        slippage_model=FixedSlippage(spread=0.0)
    set_slippage(slippage_model)
```

```
    # Used only for progress output.
    context.last_month = intial_portfolio
```

```
    # Store index membership
    context.index_members = pd.read_csv('sp500.csv',
index_col=0, parse_dates=[0])
```

```
    #Schedule rebalance monthly.
    schedule_function(
        func=rebalance,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )
```

```
def rebalance(context, data):
    # Write some progress output during the backtest
    output_progress(context)
```

```
```

```
    # Ok, let's find which stocks can be traded today.

    # First, get today's date
    today = zipline.api.get_datetime()

    # Second, get the index makeup for all days prior to
today.
    all_prior =
context.index_members.loc[context.index_members.index
< today]

    # Now let's snag the first column of the last, i.e.
latest, entry.
    latest_day = all_prior.iloc[-1,0]

    # Split the text string with tickers into a list
    list_of_tickers = latest_day.split(',')

    # Finally, get the Zipline symbols for the tickers
    todays_universe = [symbol(ticker) for ticker in
list_of_tickers]

    # There's your daily universe. But we could of
course have done this in one go.
    """

    # This line below does the same thing,
    # using the same logic to fetch today's stocks.

    todays_universe = [
        symbol(ticker) for ticker in
```

```
context.index_members.loc[context.index_members.index
< today].iloc[-1,0].split(',')
    ]
    """

    # Get historical data
    hist = data.history(todays_universe, "close",
momentum_window, "1d")

    # Make momentum ranking table
    ranking_table =
hist.apply(momentum_score).sort_values(ascending=False)

    """
    Sell Logic

    First we check if any existing position should be
sold.
    * Sell if stock is no longer part of index.
    * Sell if stock has too low momentum value.
    """
    kept_positions =
list(context.portfolio.positions.keys())
    for security in context.portfolio.positions:
        if (security not in todays_universe):
            order_target_percent(security, 0.0)
            kept_positions.remove(security)
        elif ranking_table[security] <
minimum_momentum:
            order_target_percent(security, 0.0)
            kept_positions.remove(security)
```

```
    """
    Stock Selection Logic

    Check how many stocks we are keeping from last
month.
    Fill from top of ranking list, until we reach the
    desired total number of portfolio holdings.
    """
    replacement_stocks = portfolio_size -
len(kept_positions)
    buy_list = ranking_table.loc[
        ~ranking_table.index.isin(kept_positions)]
[:replacement_stocks]

    new_portfolio = pd.concat(
        (buy_list,

ranking_table.loc[ranking_table.index.isin(kept_positions)])
    )


    """
    Calculate inverse volatility for stocks,
    and make target position weights.
    """
    vola_table =
hist[new_portfolio.index].apply(volatility)
    inv_vola_table = 1 / vola_table
    sum_inv_vola = np.sum(inv_vola_table)
    vola_target_weights = inv_vola_table / sum_inv_vola
```

```
    for security, rank in new_portfolio.iteritems():
        weight = vola_target_weights[security]
        if security in kept_positions:
            order_target_percent(security, weight)
        else:
            if ranking_table[security] >
minimum_momentum:
                order_target_percent(security, weight)

def analyze(context, perf):
```

```
    perf['max'] = perf.portfolio_value.cummax()
    perf['dd'] = (perf.portfolio_value / perf['max']) - 1
    maxdd = perf['dd'].min()
```

```
    ann_ret = (np.power((perf.portfolio_value.iloc[-1] /
perf.portfolio_value.iloc[0]),(252 / len(perf)))) - 1
```

```
    print("Annualized Return: {:.2%} Max Drawdown:
{:.2%}".format(ann_ret, maxdd))

    return
```

```
start = datetime(1997, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2018, 12, 31, 8, 15, 12, 0, pytz.UTC)
perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    analyze=analyze,
    capital_base=intial_portfolio,
    data_frequency = 'daily',
    bundle='ac_equities_db' )
```

Performance

This is where I intend to disappoint you, dear reader. Yes, you have made it all the way here, this far into the book, and now I will hold back some of the information from you. I won't give you the usual type of performance data for this model. I won't show a table with annualized return, maximum drawdown, Sharpe ratio and the usual. But I have good reasons for this. And it has nothing to do with being secretive, and everything to do with actually wanting to help you for real. Hear me out here.

If you want to learn for real, you need to do the work and crunch these numbers yourself. You need to properly understand what is going on. You need to learn every detail about how a model works, how it performs, what it's sensitive to or not, how it can be modified and a whole lot more. Simply taking my word for it, and trading the rules I described above would be a very bad idea. You wouldn't learn a thing.

I will put some graphs and tables with performance data, and you could figure out some from that. But I will refrain from the usual performance statistics, at least for now, in the hopes that you will be curious enough to try it out. After all, I'm giving you all you need in this book to replicate all this research.

This is a practical book, meant to teach you how to do things yourself. You get the full source code and you get full explanations. That's more than most trading books can claim. But this is not a book where you are supposed to simply trust the claims of the author and go follow his trading rules.

For this, and other models in the book, I will give you some idea of how it performs. I will show you the general profile of the performance and discuss the characteristics of the returns. This will give you a sufficient understanding of the model performance.

Avoiding these usual suspects of performance numbers up front also serves to reduce risks of another classic trap. If I

published these numbers, some readers would immediately start comparing. First of course, which of the models in this book performs the ‘best’. If there is such a thing. Next, how do these models stack up compared to some other book.

Neither of those comparisons would be helpful at this point. No matter who would ‘win’. Such a comparison would simply be missing the plot. Comparing models across different books by different authors, likely using very different methodology would be pointless. Comparing within the book would likely lead you to be more interested in one model than another, for all the wrong reasons.

Backtests can be geared up and down to fit the return numbers you are looking for. And keep in mind that I could easily make the numbers like annualized return higher or lower, simply by choosing which year to start and end the backtest. The general profile and characteristics of the returns are a lot more interesting, and will help you properly understand the model.

Ok, so having said all of that, you will actually get a comparison between models, including the numbers you are likely looking for, later in this book. I would just prefer not to distract you with that at the moment.

And even more importantly, this book actually does offer an excellent method of evaluating trading model performance. But that's not written by me. My fellow author and friend Robert Carver has kindly contributed a guest chapter on that topic, chapter 22.

As for now, we just need to figure out if all the hassle of this chapter was worth it or not. If there seems to be any value in this sort of approach to trading, or if we should just abandon it and move on.

Equity Momentum Model Results

Starting by taking a first glance at the monthly return numbers, as shown in

Table 12.1, it looks promising. We can see that an overwhelming amount of years are positive. Many showing very strong numbers. There are double digit negative years, which is to be expected from a long only equity model, but no disaster years.

You can also clearly see that this model, while performing quite well in the past few years, performed quite a bit better in the earlier years of this backtest. This may be an interesting topic of research, for those at home who don't mind doing some extra homework.

Table 12.1 Equity Momentum Monthly Returns

[illegible]	
19	+28.0
1998	+[illegible]
19	+42.9
2000	+[illegible]
20	-10.9
2002	-[illegible]
20	+43.0
2004	+[illegible]
20	+8.5
2006	+[illegible]
20	+13.6

2008	-38.5
20	+27.4
2010	+15.1
20	-3.6
2012	+16.0
20	+47.4
2014	+13.7
20	+-2.8
2016	+9.5
20	+17.7
2018	+-6.2

The equity curve of the backtest, shown in Figure 12-4, gives you a broad picture of how this model behaves. The top two lines show the backtest portfolio value over time as well as the S&P 500 Total Return Index, for comparison. The middle chart pane plots the drawdown, and at the bottom you will find a rolling six month correlation between the strategy and the index.

You should not make the mistake of just checking if the strategy outperforms the index. That's not the most important information here. There are other details that will tell you much more about this strategy. One thing to look at is how the

return curve levels out at times. This is when the exposure scales down, when there are not enough number of stocks available which fulfil the momentum criterion.

Second, look at the drawdown pattern. As could be expected, the drawdowns correspond quite well with known periods of market difficulties. That shouldn't be terribly surprising. After all, we are dealing with a long only equity strategy. And that leads to the bottom chart pane, which is vital to understand here.

The bottom pane shows the rolling correlation to the index. And as you see, this is usually very high. This is something that you really can't get away from, when you are building long only equity portfolio strategies. That means that it's unrealistic to expect them to perform well during times of market distress, and it's unrealistic to expect too large deviation from the market.

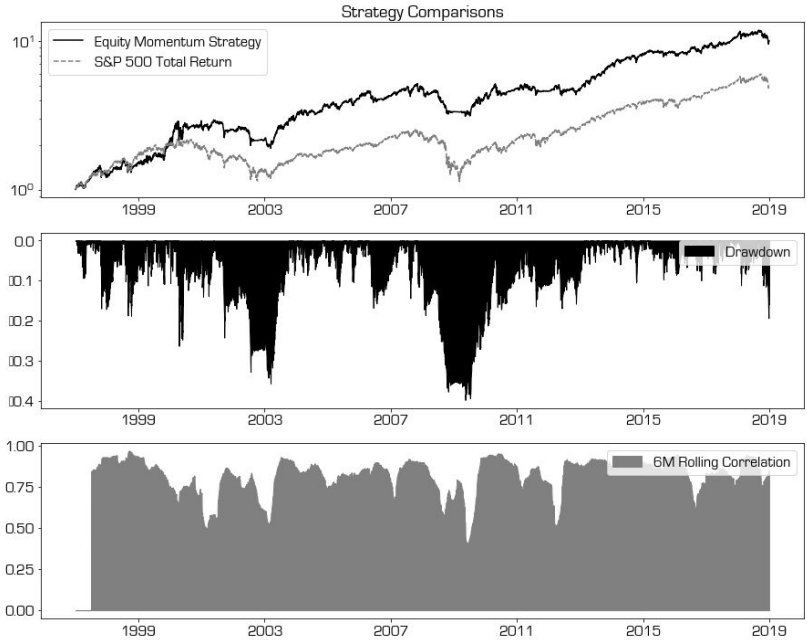


Figure 12-4 Return curve for the equity momentum model

Another way to get a feeling for long term performance of a strategy, is to make a holding period map, such as in Table 12.2. This shows what your annualized return would have been, had you had started this strategy at the start of a given year, as shown in the leftmost column, and held it for a certain number of years. The values are rounded to the nearest percent, mostly to make it fit on the pages of this book. If for instance the model had started in 1997 and traded for

10 years, it would have seen a compounded annualized gain of 16% per year. Not bad. But on the other hand, if it had started in January of 2001 and continued for 10 years, it would only have seen an annualized gain of 5%.

This type of table can help give you a different perspective on the returns than a simple graph can show. Later in this book, I will show how to calculate such a table.

Table 12.2 Holding period returns for equity momentum model

Y	22
1997	+
19	+
1999	+
20	+
2001	-
20	+
2003	+
20	+
2005	+
20	+
2007	+
20	-
2009	+
20	+

2011	+19
20	+28
2013	+32
20	+10
2015	+9
20	+16
2017	+18
20	-6

13 Futures Models

Futures are contracts to buy or sell an asset at a specified time in the future. But only a diminishingly small part of futures market participants use these contracts for such things. The original idea with futures were to enable farmers higher economic certainty by selling their future harvest at today's known prices. As with most financial innovations, it quickly turned into another part of the great casino out there and became a popular vehicle for traders.

There are some great properties of

futures that we are going to make use of in this book, and there are also some dangerous potential. The dangerous way to see futures is as easy leverage. The fact that you are able to take on enormous risk in the futures markets does not necessarily mean that it's a good idea to do so.

In the futures markets you can take positions many times larger than in the stock markets. If you have an account size of a hundred thousand, you could easily put on a position of a million or more. That's a great way to wipe out your account quickly, and this feature of the futures markets tend to appeal to the gambling type. As in a casino, you could double your money in a day or lose it all in a minute. That's not how we will use futures in this book.

This author would argue that the futures markets is the most interesting space for systematic trading, owing mostly to the vast diversification potential. The one weakness of course with futures markets is the fairly high capital requirements to play this game. The most interesting type of systematic models

require a larger capital base than most readers of this book will have available. But before you stop reading to draft a letter asking for a refund, know that almost no one makes interesting money trading their own capital. The real money is in asset management, and the fees that can be made by doing a good job managing other people's money.

Futures Basics

Futures are the only derivatives I intend to cover in this book. They are quite straightforward, easy to understand and trade. I certainly don't mean that they are simple to make money from, but they have very little in common with the highly complex world of options trading.

The one really great feature of futures is the standardization. This is the single most important thing you need to understand about futures. The very reason that they became so popular with

professional systematic traders, such as the now prolific trend following industry, is the standardization.

Because they are standardized, you can trade a very wide array or markets in the same way. You can trade cotton and lean hogs the same way as you trade equity indexes and currencies. The same mechanics, the same math, the same terminology. It makes things so much easier. There are some minor differences between futures markets, but for most purposes they work more or less the same.

The futures markets can open a whole new world for you. If you are used to being an equity trader or a forex trader, now you can suddenly trade everything in the same way. There is really very little reason to pick a single asset class to trade, when you can get so much diversification benefit from trading all of them. And the futures markets is just perfect for this.

A futures contract is mutual obligation to conduct a transaction in a specific asset at a specific future date, at a specific price. That's the textbook

definition, and while it's important to understand this part, it works a little different from a practical point of view.

The original purpose of futures is hedging. The simplest example may be a farmer who will have a crop of corn ready to ship out in three months and would like to lock in the price. He can then sell futures contracts for delivery in three months, and be sure to lock in the current price levels. A breakfast cereal company may need to buy that corn in three months, and they would also like to avoid the price risk in the next few months. They take the opposite side of that futures position, locking in current price levels.

The same example could be made with most futures markets, whether the underlying is oil, currencies or bonds, the hedging argument works in theory. Of course, in reality an overwhelming part of all futures trading is speculation, not hedging.

You buy the gold futures contract if you believe that gold is going up. If you think that the Yen is about to move down,

you go short that contract.

As mentioned, the futures contracts are a mutual obligation to take or make delivery. That's the theory, but not exactly the reality. Some futures markets have physical delivery, while others are cash settled. Commodities and currencies are generally physically settled for instance. That means that if you hold a long position of live cattle futures until they go into delivery, you may find yourself with a truck of slightly confused bovines outside your office. In theory.

These days, practically no bank or broker will allow you to hold a physical futures position past the date where it goes to delivery, unless you happen to be a commercial trader with specific arrangements allowing for physical transactions. If you have a regular brokerage account, there is no risk of unexpected bovine surprises. Your broker will shut your positions down forcefully at the last moment, if you fail to do so yourself. Not out of altruistic reasons of course. They are no more interested in dealing with physical settlement any more

than you do.

Currency futures however are usually allowed to go to delivery, so be very careful with those. If you have a currency futures contract open at expiry, you may get some larger than expected spot conversions going on in your accounts. That may or may not have happened to yours truly a long time ago, but I will strongly deny any such rumors.

Another highly important point with the standardization of futures contracts is regarding counterparty. When you buy a gold futures contract, that means that someone else out there is short the same contract. But there is no need to worry about who that person is, and whether or not your counterpart will welch on the deal or fail to pay up in time.

With futures, your counterparty is the clearing house, not the individual or organization on the other side of the trade. You have no counterparty risk towards whoever is on the other side.

Futures Mechanics and Terminology

If you are used to the stock market, the first thing you will notice with futures is the limited life span. You don't just buy gold futures. You buy a specific contract, such as the March 2017 gold futures contract, for instance. That contract will no longer exist in April of 2017, so you can't just buy and hold futures like you could with stocks. You always need to make sure that you hold a position in the right contract, and keep rolling it. The word right is dangerous of course, but most of the time the right contract would be the one that's most actively traded at the moment. For most futures markets, there is only one highly liquid and actively traded contract at any given time. Sometime before the expiry, the traders will start moving to another contract and there will be a new active contract. The process of moving positions from one month to another is referred to as rolling.

Another key thing to understand about futures is the mark-to-market procedure. When you buy a futures contract, you don't actually pay for it upfront. You do need to put up a margin, a kind of collateral, but more on that later on. As opposed to stocks and bonds, you don't actually pay the whole amount, you just need to have a small part of it available to make sure you can cover potential losses.

The mark-to-market procedure settles all positions at the end of the day. When you have futures positions open, you will see a daily cash flow. If you lost two thousand dollars on the day, that will be automatically deducted from your cash account at the end of the day. If you made a gain, it will be credited. Therefore, the value of a futures position is always zero at the end of the day. It has been marked to market.

Each futures market represents a certain amount of the underlying, not just one unit. For example, a gold contract represents 100 ounces, a crude oil contract is for 1,000 barrels and the Swiss

Franc contracts 125,000 CHF. This property is what is called contract size. For most contracts, the contract size is also the same as the more important property point value. There are a few exception to that rule, such as for bonds, but most of the time they are the same.

The reason that the property point value is more important to us is that this is what you need to multiply with to calculate your profit or loss. The point value, sometimes referred to as big point value, tells you how much money you gain or lose if the futures contract moves one full currency unit. You always need to be aware of the point value of the market you are trading.

Take gold as an example. You buy a single contract of the March 2017 gold at $1,400 on Monday. By the closing time that day, the price is $1,412, exactly 12 dollars more than the previous day. Since every full dollar move represents 100, the point value for the gold market, $1,200 will now magically appear on your cash account in the evening.

The next day, the price of the gold futures moves down to $1,408, and $400 will be deducted from your account. On the Wednesday you close the position, selling the same contract, at $1,410, and $200 dollars is credited to your account.

And that, in a nutshell, is how futures work.

Table 13.1 Futures Properties

Property	Description
Ticker	The base code of the futures contract. E.g. GC for Comex Gold. This is unfortunately not standardized and different data vendors can use different tickers for the same contract. If you use multiple market data vendors, it may be worth building your own lookup table to be able to easily translate between the different code schemes.
Month	The delivery month is expressed as a single letter, and here thankfully the nomenclature is the same for all vendors. January to December are designated, in order, by the letters F, G, H, J, K, M, N, Q, U, V, X and Z.

Year	A number to indicate which year the contract expires. Often this is just a single digit, with the decade being implied as the next possible one. For some data vendors, a double digit value is used for added clarity.
Code	The full code is the combination of the three properties above. So Comex Gold with delivery month June 2019 would usually be designated GCM9 or GCM19, depending on if single or double digit year is used.
Expiry Date	The exact date when the contract expires to either financial settlement or actual delivery. For a trader, this date is only relevant for financial futures, not for commodities or anything that is actually deliverable. For deliverable contracts you need to be out much earlier.
Last Trading Day	This is the date you need to pay attention to. The rules are different for different markets and they may use slightly different terminology for this date, (first notice day etc.) but all futures contracts have a predetermined

last day of trading for speculators. For physically deliverable contracts, you risk being forced to take or make delivery if you hold beyond this point. In practice this is not likely to happen though, as most brokers will not allow you to enter delivery and they will shut down your position forcefully on this day unless you do first. You don't want that to happen though, so you better make sure you shut down or roll your position in time.

Contract Size

This tells you what one contract represents in real world terms. The Nymex Light Crude Oil as an example represents 1,000 barrels worth, while the Swiss Franc currency future on the ICE represents 125,000 CHF.

Multiplier

For most futures contracts, the contract size and the multiplier is exactly the same. When you deal with cross asset futures though, you will run into some exceptions to this rule and that necessitates a standard way to calculate your profit and loss, risk etc. You need a way of knowing exactly how

much the profit or loss would be if the futures contract moves one full point. For bond futures the answer is usually the contract size divided by 100. With money market futures you need to both divide by 100 and adjust for the duration. So the 3 month Eurodollar future with a contract size of one million, then ends up with a multiplier of 2,500 (1,000,000 / 100 / 4). Make sure you have a proper lookup table for multiplier for all contracts you want to trade. Multiplier is also known as point value, or big point value.

Currency

For the multiplier to make sense you need to know what currency the future is traded in and then translate it to your portfolio base currency.

Initial Margin

The Initial margin is determined by the exchange and tells you exactly how much cash you need to put up as collateral for each contract of a certain future. If the position goes against you however, you need to put up more margin so you better not sail too

close to the wind here. Your broker will shut down your position if you fail to maintain enough collateral in your account.

Maintenance margin

The amount of money needed on your account to hold on to a contract. If your account drops below this amount you are required to either close the position or replenish funds in your account.

Open interest

Most financial instruments share the historical data fields open, high, low, close and volume, but the open interest is unique to derivatives. This tells you how many open contracts are currently held by market participants. Futures being a zero sum game, someone is always short what someone else is long, but each contract is counted only once.

Sector (asset class)

While there are many ways of dividing futures into sectors, I use a broad scheme in this book which makes a lot of sense for our needs. I will be dividing the futures markets into currencies, equities, rates, agricultural commodities and non-agricultural

commodities.

Every individual futures contract has a specific code. From this code, you can see the market, the delivery month and the delivery year. The standard code for a futures contract is simply those three things, in that order. The March 2017 gold, for example, would have the code GCH7, or GCH17 if double digit year convention is used.

Typically the year is designated with a single digit, but some market data providers and software vendors use double digits. In this book we will use two digits as that's the required format for the Zipline backtesting software.

When you deal with futures for a while, the syntax of the tickers will become second nature. If you are not yet overly familiar with this instrument type, you should take a moment to look at Table 13.2. Anyone trading or modeling in this area should know the monthly delivery letters.

Table 13.2 Futures Delivery Codes

Month		
January		
February		
March		
April		
May		
June		
July		
August		
September		
October		
November		
December		

Futures and Currency Exposure

When it comes to currency exposure, futures differ quite a bit from stocks. This topic is of course only relevant if you trade markets denominated in different currencies.

For stocks, you pay for your purchase up front. That means that a US based investor who buys a million Euro worth of stocks in France has to exchange his dollars for Euro. He would then have

an open currency exposure of one million, unless he chooses to hedge it.

In the case of international stocks, you would then in effect have two positions on. One in the stock, and one in the currency. When modelling and backtesting strategies, this can be a significant factor.

For futures on the other hand, the currency impact works a bit differently. Remember that we are not paying for futures contracts the way we pay for stocks. We just put up a margin, and then have the daily profit or loss netted on our account.

That means that your currency exposure for international futures is limited to the profit and loss.

If you buy a British futures contract, you have no initial currency exposure. If by the end of the first day, you make a thousand Pound profit, that amount will be paid to your Sterling account. Now you have an open currency exposure of one thousand, but not on the full notional amount of the contract. If you lose the

same amount next day, now you have no currency exposure.

Thereby, currency exposure is a much smaller issue with futures, and one which can almost be approximated away.

Futures and Leverage

The fact that we only need to put up a margin, and not pay for futures in full, can have some interesting implications. This means that in theory, we can take on very large positions.

It's rarely worth wondering just how large positions you could take on, if you were to max out your risk. You can take on far larger positions than you ever should anyhow. And that possibility is often misused by retail traders. If your motivation to trade futures is that you can take on higher risk than with stocks, you should stay with the stocks. Using futures for cheap leverage is a very bad idea.

But the possibility of taking on these enormous position sizes has a very clear advantage. When you deal with cash

markets, like stocks, you are more or less limited to 100% notional exposure. Sure you can leverage stocks, but that's expensive and restrictive.

But with futures, you can start by deciding what risk level you are looking for, and use that to guide your position sizes. There is no need to really care about the notional exposure of your portfolio, as long as you have a firm eye on the risk side.

This means, among other things, that we are able to take on meaningful positions in very slow moving markets. The short term interest market for instance, where daily moves larger than 0.1% are rare, wouldn't be of much interest if we were limited to 100% notional exposure. It just wouldn't be able to have a noticeable impact of the portfolio.

But this is a clear benefit of futures. We can use risk as a guide, instead of notional exposure.

Leverage can wipe you out, but correctly and deliberately used, it allows

you to construct much more interesting trading models.

14 Futures Modeling and Backtesting

Backtesting strategies for equities and for futures are two very different things. Even if your high level, conceptual trading rules are similar, the model logic and code are by necessity not. The most important difference comes from the fact that futures contracts have a limited life span. This has multiple implications. The most obvious one being that we have to keep track of which exact contract delivery to trade, and when to switch to another one. But that's just scratching the surface of this issue.

Technically speaking, we don't have long term time series in futures world. Any given futures contract has a limited

life span and only has sufficient liquidity to matter for a few months at the most. Quantitative trading models however usually need to rely on a substantially longer time series to analyze. Take something as simple as a long-term moving average. It just is not possible to calculate, using actual market data.

As an example, look at Figure 14-1, which shows the price and open interest of the February 2003 crude oil futures contract. Remember that open interest tells you how many futures contracts are open at the moment. When you buy a contract, that number goes up by one. When you sell it again, the number goes down by one. This is a good indication of how liquid a contract is.

This contract started trading, or at least was possible to trade from the fall of 2000, but as you can see in the graph, it really had no trading to speak of until late 2002. Even though we have pricing for the contract for years, most of that data is irrelevant. As it was not traded in any meaningful volume, we can't rely on such a price series or draw any conclusions at

all from it. The historical data for the contract only becomes interesting when there is enough market participation to ensure proper pricing and sufficient liquidity for actual trading.

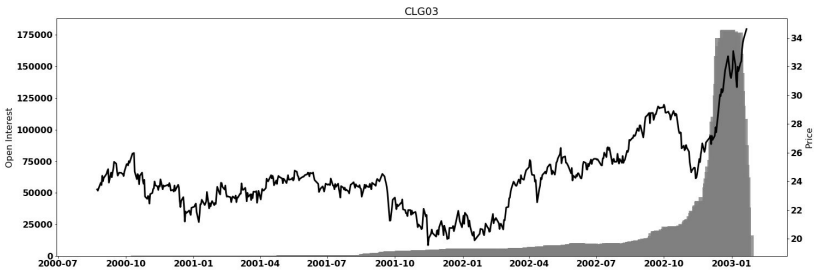


Figure 14-1 Single Contract - CLG03

As you see in Figure 14-1 the contract, February 2003 crude oil, is only really relevant for a very brief period of time. That's how it normally looks. In the next graph, Figure 14-2, you should get a better view of what is really going on here. This shows five consecutive deliveries of crude oil, for the same time period. Here you can see how the liquidity moves forward, from contract to contract, over time. In the bottom chart pane, you see how the October 2002 holds the most liquidity, but how it slowly gives ground to the November contract. After a month, the November contract fades, and

December takes over. This circle will keep repeating, at a fairly predictable pattern.

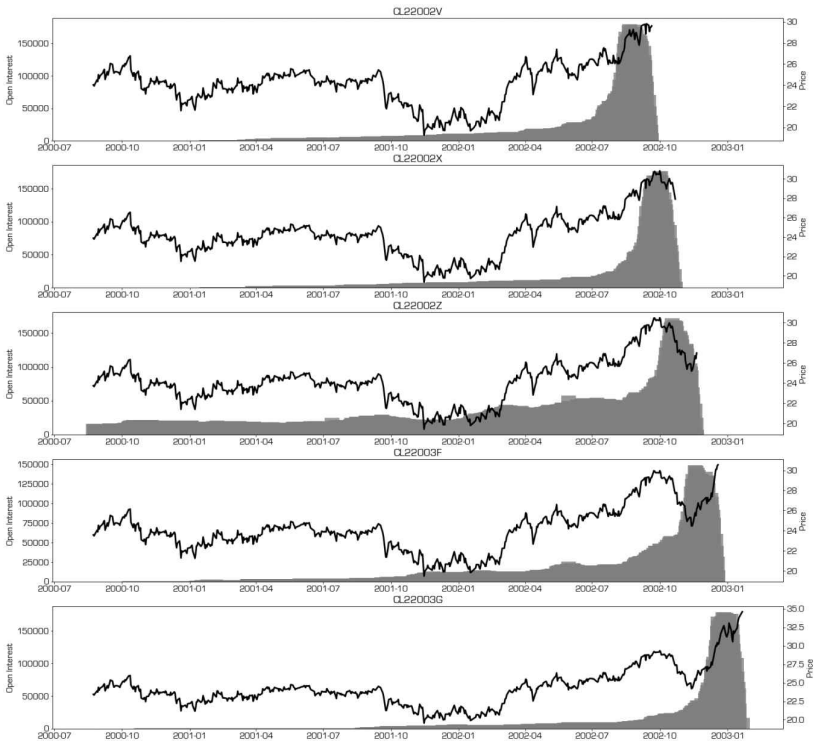


Figure 14-2 Liquidity Moving from Contract to Contract

It's important to understand that, from an analytics point of view, that each contract only really matters during that brief time that it's the king of the hill, the one with the highest liquidity. That leaves us with a whole lot of short time series, from different contracts, and no long term history.

A surprisingly common rookie mistake is to make a long term time series by simply putting one contract after another. Just using the same time series as they are, and switching between contracts as we go. The problem with this is that this method will create a flawed representation of what actually happened in the market. It will plain and simply be wrong.

The price of November crude oil and the December crude will be different. It's simply not the same thing, and can't be compared on the same basis. One is for delivery in November and the other for delivery in December. The fair price for these two contracts will always be different.

In Figure 14-3 you can see how the same underlying asset, in this case still crude oil, is priced differently on the same day, depending on the delivery date. This kind of difference is normal, and it has real world reasons. It has to do with cost of carry, as this represents the hedging cost, and thereby fair pricing. Financing cost and storage cost are major factors

here.

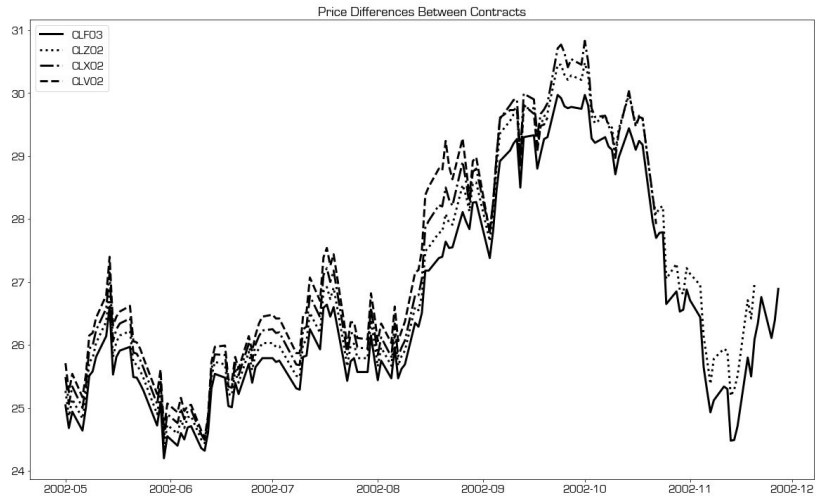


Figure 14-3 Comparing Contracts

Some market data vendors still provide time series where the contracts are just put one after another, without any adjustments. Even some of the extremely overpriced market data vendors, those charging around two thousand bucks a month, do this. Just because someone sells you data at a high cost, does not mean that the data is accurate or useful.

The effect of using a long term time series of contracts simply placed after each other, without adjustment, is the introduction of false gaps. As this so called continuation switches from one

contract to another, it will appear as if the market jumped up or down. But no such move actually occurred. If you had been long the first contract, sold it and rolled into the next contract, you wouldn't have experienced the gain or loss of a gap that this flawed continuation would imply.

So be very careful with your data when it comes to long term time series for futures.

Continuations

The most commonly used method of constructing long term futures continuations is to back adjust. What that means is that each time you roll to a new contract, you adjust the entire series back in time. Usually this is done by preserving the ratios. That's to make the percentage moves of the continuation reflect actual percentage moves.

The point with this method is to arrive at a time series which roughly reflects the experience that a real life trader would have had if he had bought

and rolled the position, month after month and year after year. Such a continuation is used for long term price analysis, but clearly can't be traded by itself. It's not a real series, it's just constructed for analytical purposes. You can't run a 200 day moving average on a single contract, but you can do it on a properly constructed continuation.

What most backtesting software will do, is to trade this artificially constructed continuation and pretend that it's a real asset. That has been the go-to solution to the futures problem on the retail side for decades, and it's also quite common on the professional side. While that's a good enough solution for most tasks, it does have some drawbacks.

One issue with this standard approach is that the longer you go back in time, the more the calculated continuation will diverge from the actual market price. If you keep tacking on adjustment after adjustment for a long period of time, you will end up with a price quite far from the market price at the time.

The regular method of making such a continuation means that the entire series is calculated on the day when you run your backtest. That often results in the kind of discrepancies that we see in Figure 14-4. The solid line on top, the one that spikes up to nearly 700, is the back adjusted series, while the lower one, that never moves much above 100 is the unadjusted series. Neither of these two series can be said to be right, just like neither is really wrong. Depending on your point of view.

The adjusted series is meant to reflect percentage moves that would have been experienced by a trader or investor holding a long exposure over time. But clearly, the price of oil was never up around the 700 mark. This distortion is the effect of adjusting for the basis gap over many years.

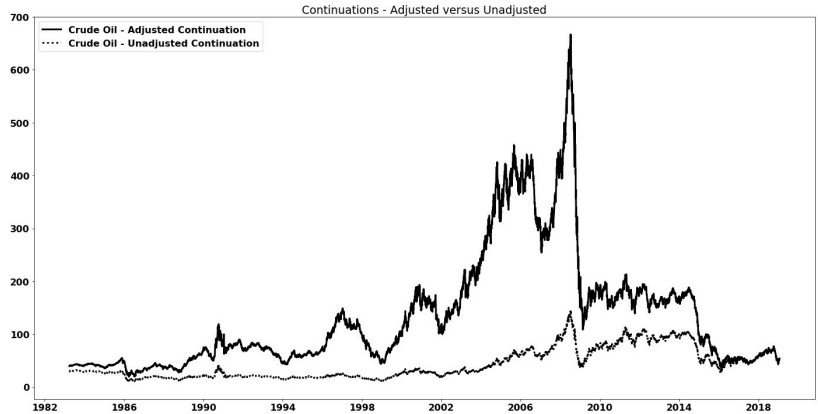


Figure 14-4 Adjusted versus Unadjusted Continuation

If the purpose with the continuation is to measure the trend for instance, that wouldn't pose much of an issue. This type of purpose is the reason for making a continuation. But the way most retail backtesting software works, the program will assume that the continuation is an actual tradable instrument. It will buy and sell the continuation itself, which clearly is not possible in real life.

That leads to some potential issues. Crude oil futures has a multiplier of 1000, meaning that a price move of one dollar will impact your mark-to-market valuation with 1,000 dollars. This is something that's normally taken into

account when deciding position sizes, and can lead some potentially flawed conclusions for smaller portfolios.

Assume for a moment that your backtest is running a portfolio of one million dollars, and that you are targeting a daily variation per position of 0.2%. That is, you want each position to have an approximate daily impact on your portfolio of $2,000.

The backtest is trying to buy a position in oil, which as mentioned has a point value, or multiplier of 1,000, and which has an average daily price variation of about one percent. Running your backtest on the adjusted series some years back, the adjusted oil price was $400, and tends to vary by about $4 per day.

As the contract has a multiplier of 1,000, each contract fluctuates by a value of $4,000 per day. But we wanted to take a position that fluctuates by $2,000 per day, and therefore the conclusion is that we are unable to trade. It's simply not granular enough.

But if we look at the actual contract

at the time, it was traded around $40, in unadjusted terms. With a daily change of about one percent, that means that each contract fluctuates by about $400 per day, and we are able to buy 5 whole contracts.

This is something that you have to be careful with. You could usually find ways to compensate for such issues, but it does mean that you need keep in mind what kind of distortions are possible and find ways to address them.

Another problematic aspect of using such continuations is that they are not static. The traditional way of using continuation involves recalculating the entire series, adjusting all points back in time when we roll to a new contract. Usually this is done by applying a ratio to the entire existing time series. Stop and think about the implications of that for a moment.

It can have some rather worrying implications. Consider a long term trading model that you constructed a year ago. Back then, you ran a complete backtest of the model, and you were happy with the

results. Now, a year later, you run another backtest of the same rules on the same markets. And you find that the results are not the same anymore. This can very well happen, again depending on what analytics you apply on the time series, and whether or not you have taken this phenomenon into account.

A fellow market practitioner who was kind enough to give early comments and suggestions for this book pointed out that I'm being too harsh on the use of continuations, and he is probably right. If you really know what you are doing, and fully understand how your continuations are calculated and the implications of those calculations, you can make sure that your model code does not do anything silly. But I would still maintain that you can achieve a higher level of realism using individual contracts, or with dynamically calculated continuations. And going forward in this book, we will use both.

Zipline Continuation

Behavior

If nothing else, the way Zipline handles futures is reason enough to choose this backtester over others. I would like to take some credit for this, as I advised Quantopian a few years ago on how to best handle futures, but I suspect that they were merely humoring me and that they already had plans in this direction.

Their Zipline backtesting engine has a neat solution to the age old continuation issue. The Zipline continuations are generated on the fly, on each day of the backtest. That means that the continuation will look as it would have, if calculated on that day. At each day the backtest processes, the continuations will be freshly generated from that day, so that the 'current' prices in the continuation reflect actual current prices in the market.

The regular, or perhaps we should call it the old school way of calculating a continuation is done in advance, usually as of the real life time that the backtest is

performed. Zipline has a clever solution to this, by calculating a continuation each time it's requested in the backtest, based only on data available up until that point in time. The results are more realistic and less prone to errors.

Zipline also solves another common issue relating to continuations. As mentioned, most backtesting solutions available to retail traders will pretend to trade the continuation itself. That's clearly not possible in the real world, as the continuation is no more a real tradable instrument than a moving average. There can be some unexpected implications of simulating trading on the continuation itself, and this can have a real impact on the realism and accuracy of your backtest if you are not careful.

Zipline solves this by trading only actual contracts, just like you would in reality. Clearly this is the better solution, but is also means that you need to provide the backtesting engine with historical data for each individual contract. It could end up to be quite a few of them. For the purposes of this book, I used a database of

around twenty thousand individual contracts.

This way of working with futures is far superior to what is normally available to retail traders. It's not only more realistic, but it opens up for brand new possibilities such as calendar spreads or term structure based trading. Yes, I'm just going to throw those words out there and let you think about it for a while.

The flipside is of course that it gets a little more complicated. We need to feed substantially more data to the backtester. Not just the historical time series data, but also metadata. For each contract, we need to know things like expiry date, last trading date and more.

When building a backtest, we need to keep track of which contract to trade at any given time. And of course, we need to keep track of when it's time to roll to the next contract.

It may sound much more complicated, but it's really not that difficult to set up.

Contracts, Continuations and Rolling

From the equity section of this book, you should already be familiar with how to access data in a Zipline backtest. There are few new things in this regard when it comes to futures, and it has to do with the fact that in futures land, we have to deal with both continuations and individual contracts.

A continuation can be created easily, by providing a root symbol and a few basic settings. The line below shows how to create a continuation for the big S&P 500 futures, root symbol SP.

```
sp_continuation = continuous_future('SP', offset=0, roll='volume', adjustment='mul')
```

With this continuation object, we can then request historical time series, check which contract is currently active, and we can even pull the complete chain of traded contracts, as we will do in later

chapters.

You can request the time series history the same way as we did earlier for stocks.

```
continuation_hist = data.history(
    sp_continuation,
    fields = ['open','high','low','close'],
    frequency = '1d',
    bar_count = 100,
)
```

This time series is what would typically be used for analytics. Here we get the adjusted series back in time, using only data available at the time the backtest made the request.

But to be able to trade, we need to get hold of an actual, individual contract. Most, but certainly not all, futures models trade the most active contract. If that's the case, we can just ask the continuation which underlying contract it's using at the moment.

```
contract = data.current(cont, 'contract')
```

Once we have a contract, placing orders works the same way as we saw earlier for stocks. We can order a set

number of contracts, target a fixed notional amount, or target a certain percent exposure.

```
order(contract, 100)  # Buys 100 contracts
order_value(contract, 1000000) # Buys a million dollar notional, or closest.
order_target(contract, 10000) # Adjusts current position, if any to 10,000 contracts
order_target_value(contract, 1000000) # Adjust to target a million dollar notional.
order_target_percent(contract, 0.2) # Adjusts current position to target 20% exposure
```

As mentioned previously, the hassle with trading individual contracts is that they have a short life span. It's not just a matter of deciding which one to pick when we enter the position. We also have to keep track of when the contract that we hold is no longer active. At that time, we need to roll to the next contract.

If the intention is to stay with the most active contract, the easiest way is to run a daily check, comparing currently held contracts with the ones that the continuations use at the moment.

```
def roll_futures(context, data):
    open_orders = zipline.api.get_open_orders()
```

```
    for held_contract in context.portfolio.positions:
        # don't roll positions that are set to change by
core logic
        if held_contract in open_orders:
            continue
```

```
        # Make a continuation
        continuation = continuous_future(
                held_contract.root_symbol,
                offset=0,
                roll='volume',
                adjustment='mul'
                )
```

```
        # Get the current contract of the continuation
        continuation_contract =
data.current(continuation, 'contract')
```

```
        if continuation_contract != held_contract:
            # Check how many contracts we hold
            pos_size =
context.portfolio.positions[held_contract].amount
            # Close current position
            order_target_percent(held_contract, 0.0)
            # Open new position
            order_target(continuation_contract, pos_size)
```

As you will see in the futures models, we will use this code or variations thereof to ensure that we keep rolling the contracts when needed, and stay with the

active one.

15 Futures Trend Following

Some years ago I wrote a book called *Following the Trend* (Clenow, Following the Trend, 2013). In that book I presented a rather standard trend following model, which I used to explain what trend following is and how it works. I demonstrated that the bulk of the returns from the managed futures industry can be explained by a very simple trend following model. In retrospect, my only regret is not making the demonstration model even simpler.

I wanted to include various features in the model to demonstrate how they work and what impact they can have. Had I kept the model rules even simpler, I might have been able to avoid the occasional misunderstanding that this

model was some sort of *Secret Trend Following Super System* .

The rules I presented were not exactly secret, as they had been known for decades prior to my book. What I attempted to do was to explain it in a hopefully new way and give people outside of our industry a feeling for what it is that we do for a living. There are far too many inaccurate myths around the business.

Later in this book, I will correct that mistake by presenting an incredibly simple trend following model. One that rightfully should shock people. It sounds so dumb that most would dismiss it out of hand. But it still shows pretty good returns in a backtest.

But I would assume that people who read my first book want to know what is going on with the Core Model, as I called it in that book. It's fair to ask if those rules have worked out, or if they crashed and burned. The short story is that the approach did just fine.

I suspect that many of the readers

of this book also read the first one. Still, I don't want to assume that all of you did, so I will be sure to include enough information here that you are not forced to go buy the other book. I'm not going to repeat the entire thing of course, but I will try to repeat as much as is needed to follow the lessons of this book.

The model from *Following the Trend* had quite an interesting ride. As it turns out, my book was published just as a few rough years for trend following started. The last year covered in the book was 2011, which was a generally poor year for trend following strategies. Of course, 2012 and 2013 were also quite poor years for trend models.

Every time trend following has a bad year or two, there are pundits coming out of the woodwork to declare the strategy dead. It really does not matter how many times this happens and how many times they are wrong.

The thing is that trend following, like every other strategy, has good years and bad years. If you find a strategy that

has not had a poor run, there is probably something that you overlooked. The only person who never had trading losses is Bernie M.

But then came the great 2014, when trend following did really well. The losses of the past three years wiped out and brand new all-time highs made. It was a really good year and the people who called the end of trend following were nowhere to be seen.

With trend following, you always need to keep in mind that it's a long term strategy. You might see several years in a row of losses, just as you will in most strategies. This is the reason why you should trade in reasonable size. Sooner or later, any strategy will have a couple of consecutive losing years.

Every time trend following has a few bad years, a lot of players get shaken out. Those who stay the course have always been rewarded in the long run.

Principles of Trend

Following

The idea of trend following is really quite simple. Most robust trading ideas are quite simple. This of course does not mean that it's simple to implement, simple to make money from or even necessarily simple to code. Simple, means that the idea itself is simple. The concept is easily understood and explained.

This is something worth considering when you design a strategy. If you are unable to explain the idea behind your trading strategy in a simple, brief and understandable manner, then there is a clear risk that you have overcomplicated and over fitted rules to match data, and that there is little to no predictive value.

Trend following is based on the empirical observation that prices often move in the same direction for a sustained period of time. The strategy aims to capture the bulk of such moves, while not attempting in any way to time valleys or peaks. It simply waits for prices to start moving in a certain direction, and then

jumps onboard in the same direction, attempting to ride the trend. Most commonly, a trailing stop loss is employed, meaning that we wait until the market starts moving against us before exiting.

Trend following trades fail most of the time. But that's ok. Trend following tends to have a fairly large amount of losing trades, often as high as 70 percent. It's not a strategy for those who like to be proven right all the time. What is important however is the long term value development of the portfolio as a whole. Trend following tends to have a large amount of small losses, and a small amount of large gains. As long as the net expected value of this works out to a positive number, we are all good.

There are myriads of methods to choose from when designing and implementing a trend following model. If you get the fifty best trend following traders together to spill their secrets, most likely they will have fifty very different set of rules. But as I demonstrated in *Following the Trend* , the bulk of trend

following returns can be explained by very simple rules.

In that book, I showed how an extremely simple rule set has very high correlation to the returns of the world's leading trend following hedge funds. I did not do that to diminish the impressive work of any of these people. I did it to explain trend following as a phenomenon, and educate readers on where the returns come from and potentially how to go about replicating it.

Trend following can in practice be a very frustrating strategy to trade. You will always enter late into a breakout, buying after the price has already rallied. Quite often, the price simply falls right back down after you enter, resulting in a swift loss. When the price does take off and trend for you, the trailing stop logic ensures that you will always lose money from the peak reading of your position, until the stop is hit. And of course, sometimes, trend following simply does not work very well for a year, or even several years.

Well, all of that is just part of the cost of doing business in the trend following field. It's not for everyone. It's not an amazing no-lose, get-rich-quick strategy that everyone should bet all of their money on. But it is a strategy which has empirically performed remarkably well for decades.

Revisiting the Core Trend Model

The Core model that I presented some years ago in *Following the Trend* aims to capture medium term trends across all market sectors. It's a very deliberate middle-of-the-road type of model. There is nothing particularly remarkable about this model. The fact that it's not only profitable, but quite handsomely profitable is what is remarkable. The fact that the return stream is very highly correlated to the trend following hedge funds of the world is what should catch your attention here.

What this should tell you is that the bulk of trend following returns can be captured with quite simplistic models. Constructing a simulation that performs similarly to some of the world's best hedge funds is not all that hard. Implementing is of course a different story, but for now we are focusing on the modelling.

Constructing a basic trend model like this is a good exercise, in part to learn how to write the code but even more importantly to understand the core logic of trend following.

Model Purpose

The idea with the core trend model is to guarantee participation in any medium to long term trend across all the major futures sectors.

There has been much mystique around trend following and a great deal of people without industry background were making money selling system rules to gullible retail traders. I wanted to

introduce you to the man behind the curtain.

But don't mistake the simplicity of the rules for a recipe to get rich quick. Understanding trend following and successfully implementing it are two very different things. For one thing, the proverbial devil tends to be in the details. While you can replicate the bulk of trend following returns with simple rules, you may require added complexity to control volatility, maintain acceptable risk and to develop a strategy that can be successfully marketed and funded.

Then of course there is the pesky little detail of capital requirements. In *Following the Trend* , I argued that you need an account size of at least a million dollars to implement a diversified trend following model. That's probably the sentence in that book that generated the most emails. I can't even begin to count how many people sent me email asking if it could be done with a much smaller account size.

I will be generous here and say that

perhaps you can do it on half that account size. Perhaps. I will even concede that you could, in theory, do it on a lot less by simply pretending to have the money.

Yes, pretending. When you trade futures models, you don't actually need to put up all of the cash, as you would for equity trading. So in theory, you could have an account of a hundred thousand dollars and simply pretend that you have a million. Trade it as if the base account size is a full million.

But that would be a very bad idea. A drawdown of ten percent would wipe you out.

This is the problem with trading diversified futures models. That they require quite a large account size. This stems from the fact that most futures contracts are quite large. While you can buy a single stock for a few bucks, the notional exposure on a futures contract can be in the tens or hundreds of thousands.

Even if you don't currently have a million dollars on your account, or have

realistic plans of acquiring it in the next few hours, you can still benefit from understanding trend following. So please don't stop reading just yet. After all, you have already paid for this book.

Investment Universe

All else being equal, a large investment universe usually helps the end results. It can be difficult to trade and manage a large universe, but in practice it's the best way to go. That's why you need to take great care when you start cutting down the universe to fit what you are able to trade.

We will use about forty markets in this demonstration. This investment universe covers agricultural commodities, metals, energies, equities, currencies, fixed income and money markets. I will use only USD denominated markets. Not because that's somehow better, because it really is not. Clearly there are more interesting markets trading in USD than in any other currency, but there is a clear

benefit in including international markets. It can greatly help with diversification and improve long term results.

No, the reason for only using dollar based futures markets here is that the backtester used for demonstration in this book, Zipline, does not yet support international futures. There is no mechanism to take the currency effects into account.

With futures this is much less of an issue than with stocks. When dealing with international futures, you don't have an open currency exposure on the notional amount. The face value of your contract in a foreign currency is not impacting your currency exposure. Remember that you haven't actually paid for this contract, as you do with stocks. You just have a margin requirement.

Trading Frequency

The model operates on daily data and checks every day for trading signals. There may not be anything to trade each

day, but we check to make sure on a daily basis. No action is taken intraday, and no stop orders are placed in the market. All logic operates on closing prices, and trading always occurs a day after the signal.

Note that two separate tasks are performed each day. First we check if any position should be closed or opened. After that, we check if any position should be rolled. The roll logic and reason for it are all explained in chapter 14.

Position Allocation

The position sizes taken for this model aim to take an equal amount of risk per position. And no, risk has nothing at all to do with how much money you lose if your stop loss point is triggered. That's unfortunately a common explanation of risk found in many trading books, but it has really nothing to do with risk as the word is used in the finance industry.

Risk is somewhat controversial word inside the business as well. In this

context I'm using a slightly simplified but still valid definition, based on expected average daily profit or loss impact.

Since we have really no reason to believe that one position is more important or will be more profitable than another, this model aims to put more or less the same risk in each position. The first and obvious thing to point out is that we can't simply buy an equal amount of each. That does not even work for reasonably similar instruments like stocks. Futures can show extreme differences in how they move, and if you hold an equal amount of each you would have a highly skewed risk.

For some markets, like oil for instance, a 3% moves in a day happens quite often. These days, you barely see a headline in the news about such a move. It's just too common. But for the US 10 year treasury futures to move 3% in a day, it would take a cataclysmic event. If you placed $100,000 in each, you would clearly have significantly more risk on the oil position than the treasury position.

A common method of solving this is to look at recent volatility for each market, and scale positions based on that. Daily volatility being a far proxy for risk in this context.

What we want to achieve is to have an approximate equal daily impact on the overall portfolio from each position. We are simply trying to give every position an equal vote. The same ability to impact the overall portfolio profit and loss bottom line.

Volatility can be measured in many ways. In *Following the Trend* , I used Average True Range (ATR) as a measurement of how much a market tends to move up and down on an average day. As I'm always in favor of expanding one's toolkit, I'm going to use a different measurement here.

Please don't email me to ask which is best, because that would be missing the point. I'm not the guy who will give you exact instructions on how to do things the best way. Mostly because I find that type of advice detrimental, at best. I want to

help you built a skill set and an understanding of various types of methods and tools. A foundation which you can build upon and actually learn how methods work, how to incorporate them into your trading ideas and hopefully how to construct your own tools.

Instead of using ATR, this time around I'm going to use standard deviation. Not the usual annualized standard deviation in percent, but rather standard deviation of price changes. It's a measurement not all that different from ATR. As it uses only closing prices, we will have a little less data to bother with, and a little simpler logic. The standard deviation here is calculated based on the daily dollar and cent changes from day to day, over a span of 40 days. Note that we are talking about price changes, not price. Using a standard deviation of the actual price levels would produce a rather nonsensical result in this context. 40 days here, roughly measures the past two months' volatility. Feel free to experiment with other settings.

Lucky for us, calculating standard

deviation of price changes is really simple in Python. You can do it in a single line, if you like. The code below is all that you need to calculate the standard deviation of price changes for the past two months, assuming you have a DataFram e with a column called clos e .

```
std_dev = df.close.diff()[-40:].std()
```

Let's break that row down to its parts, to make sure you understand what is going on here. The variable d f here is a **DataFrame** with a column called clos e . The first part of the statement simply gets the day by day changes, by using the function diff( ) . That by itself just gives us another time series, with the difference between each day.

After that, we slice off the last 40 days, and request a standard deviation calculation based on that, with the built in function .std( ) . Now try replicating this one line of code in your other favorite programming language, and you will quickly see just how much simpler Python can be.

Suppose that we would like each

position in the portfolio to have an average daily impact equivalent to 0.2% of the entire portfolio value. The number 0.2% in this example, is referred to as a risk factor. This is a completely arbitrary number which we can toggle up or down to increase or decrease risk. It's our primary risk guidance tool for the demo model in this chapter.

Here is the formula we would apply to calculate the volatility based position size.

$$\text{Contracts} = \frac{\text{RiskFactor x TotalPortfolioValue}}{\text{StdDev x PointValue}}$$

This formula assumes that everything is in the same currency. If you are dealing with international futures, you also need to translate foreign currencies to your base currency.

Suppose that we have a portfolio with a total balance of $5,000,000 and we would like to apply a risk factor of 20 basis points. The expression basis points in this context refers to fractions of a percent, so 20 basis points is the same as 0.2%. We are now about to buy crude oil, which in our example is currently trading

at $43.62 and has a 40 day standard deviation of $1.63. Here is our formula.

$$RiskFactor = 0.002$$
$$TotalPortfolioValue = 5{,}000{,}000$$
$$StdDev_{40} = 1.63$$
$$PointValue = 1{,}000$$
$$TargetImpact = 0.002 \times 5{,}000{,}000 = 10{,}000$$
$$AverageMove = 1.63 \times 1{,}000 = 1{,}630$$
$$\frac{10{,}000}{1{,}630} = 6.13$$
$$Contracts = 6$$

In this case, we end up buying only 6 contracts, for a portfolio of 5 million dollars. Well, I already warned you that this might not be the exact rules you will be implementing. Stay with me. This model is meant to teach you how things work.

What is key at the moment is that you understand the logic behind this type of position sizing. The idea being that in this case, we want every position to have a daily average impact on the portfolio of about $10,000, and from there we will figure out how many contracts we need to trade.

There are alternative ways to do volatility parity allocation, which is the fancy pants name for what we are really doing here. The other methods you may have in mind are probably fine a well. But

the concept is important.

The code for the position size calculation can be found below. Inputs are current portfolio mark-to-market value, standard deviation and point value. The risk factor is already defined in the settings section of the model, as you'll see in the complete code later on. Output is number of contracts that we should trade.

```
def position_size(portfolio_value, std, point_value):
    target_variation = portfolio_value * risk_factor
    contract_variation = std * point_value
    contracts = target_variation / contract_variation
    return int(np.nan_to_num(contracts))Trend Filter
```

A good old dual moving average is used for this model, but not for trend signals. Instead, this is used as a filter to determine the trend direction.

We are using a combination of a 40 day exponential moving average and a longer one of 80 days. If the 40 day average is higher than the 80 day average, we determine that the trend direction is positive. If it's the other way around, with the faster EMA lower than the slower, we will call the trend negative.

This by itself does not trigger any trades. We will just use it as a filter. We will only take long entry signals if the trend is positive, and conversely we will only take short entry signals if the trend is negative.

In case you are wondering why the numbers 40 and 80 were selected, the answer is, perhaps not too unexpectedly at this point that they were picked rather frivolously. These numbers are reasonable, as are many others. Feel free to try other combinations. Just keep in mind the general purpose, of determining if the market is moving up or down at the moment.

You may also wonder why an exponential moving average was selected, instead of a regular simple moving average. The primary motivation for selecting that here is to show you how to an exponential moving average is calculated using Python.

Figure 15-1 demonstrate the idea of this trend filter. As the faster EMA is higher, the trend is deemed to be positive,

else negative.

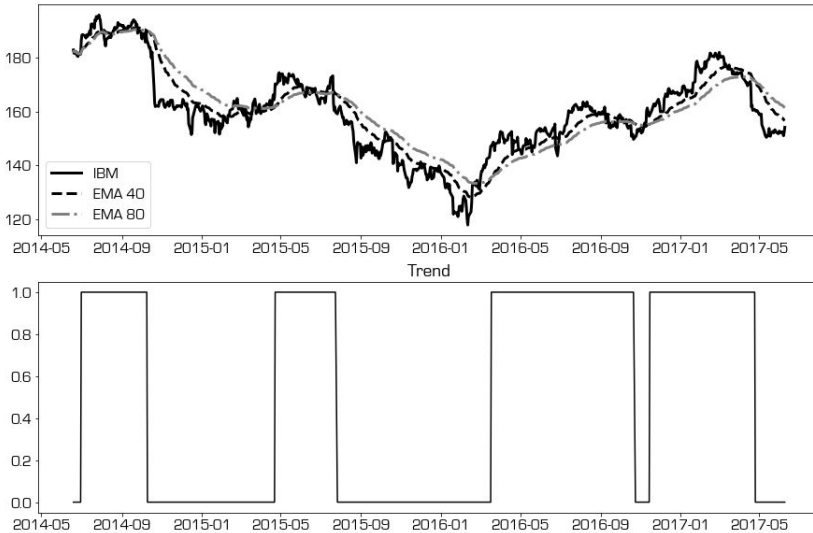


Figure 15-1 Trend Filter

Entry Rules

The trading rules for this model are symmetrical. That means that we are treating the long and short side equally, simply reversing the sign for the trading logic.

That might not be the best way to trade. Bullish trends and bearish trends tend to behave quite differently and may require different parameter sets. But for now, we will keep the model simple and

straight forward.

Entry rules are based on simple breakout logic. When a market hits a new 50 day extreme in the direction of the trend, we will enter.

For a long entry, we first require the trend filter to be positive. The 40 day exponential average must be higher than the 80 day equivalent. That means that we have a green light to take long entry signals if and when they come around.

When the price makes a new 50 day high, we enter long on the following day.

To figure out the short entries, simply reverse the logic. If the trend filter is negative, and we get a new 50 day low, we go short.

Exit Rules

This simple trend model uses a trailing stop. That means that the stop moves along with the price, making sure that we don't give back too much after a good run. What we want to do here is aim

for a certain amount of giveback of profit.

Keep in mind that we are dealing with a trend following model here. Trend models always give back part of the profit before closing. These models don't aim at buying at the bottom and selling at the top. That's not what trend following is about. Other types of strategies may attempt that, but what we are doing here is quite different. We are looking to buy high and sell higher. Not to buy low and sell high.

For this model, we have no profit target. We want to be in the position as long as possible. We simply stay in the position as long as it keeps moving in our direction. If it makes a significant move against us, we close it down. Naturally, the key word here is *significant* .

Using a percentage loss on the position is not going to help you. Remember that we are dealing with many different markets across many asset classes. The base volatility level will be quite different so you can't compare percentages.

For oil and platinum, a 2% move is not a big deal. It happens all the time. But for the US 2 year treasury futures, that's a huge move. For money markets, it's practically unheard of. No, we need something better than percentages to measure moves. We need to normalize the moves to volatility, so that they can be compared.

Earlier we discussed standard deviation in the context of position sizing. That measurement works fine for use in this context as well. What the standard deviation tells us is approximately how much a market tends to move in a day, on average.

The exit rule here is based on this standard deviation value. From the peak reading of the position, that's at the time of the largest unrealized gain, we let the position lose three times the standard deviation value before closing it.

This is not an intraday stop, placed in the markets. This is a trigger based on daily prices, with the actual trade executed the day after. This just to keep

the rules simple and easy to replicate. Most of you can probably source daily data with little to no problem, whereas historical intraday series can get expensive.

There is an added benefit of using a stop level based on standard deviation in this case. Earlier in the example, we used a risk factor of 20 basis points to set the position size. The formula, as explained earlier aims to make the average daily variation of the position impact the portfolio with about 20 basis points.

So if the market moves by one standard deviation, the overall portfolio impact will be 0.2%. And since we now set the stop at three times the standard deviation value, we now know that every position will lose about 0.6% of the portfolio value before it's closed out. That is, lose 60 basis points from the peak reading. That's the amount of profit we will give back before a position is closed, if there was a profit to give back.

Costs and Slippage

Zipline has quite a complex built-in modeling of cost and slippage, much more so than even most commercial grade solutions. If you dig into the details, you will find a range of highly configurable algorithms for taking these important variables into account. I very much encourage you to try variations, model different assumptions and see how it impacts your results.

For this demo model, I have chosen a slippage model which takes traded volume into account. It will, in this case, make sure that we never trade more than 20% of the daily volume, and model slippage based on that. Of course, this slippage model works even better on intraday data.

For costs, I have set an assumed commission per contract of 0.85 dollar, as well as an exchange fee of 1.5 dollar. Assuming you trade with a low cost online broker, these cost assumptions should be reasonably realistic.

I have left settings in the code, so that you can turn on or off slippage and

commissions and experiment with how much of an impact these things have.

Interest on Liquidity

As you should be aware by now, futures are traded on margin. Most professional futures managers tend to hang out in the 10-20 percent margin-to-equity range, and that means that some 80-90 percent of your capital is not actually needed at the moment.

Professional futures managers use that excess cash to buy short term government paper. There are two reasons for this. The most obvious is that it can earn an interest, while being the very definition of risk free. In the good old days of futures trading, the 80s, 90s and even part of the 00's, this interest could have a substantial impact on the result of the strategy.

And of course, the best part here is that as a hedgie, you get paid a performance fee on the return you generate for your clients. Even if part of

that return came from risk free interest. Yep, those were the days.

Unfortunately, these days you get so little return on these instruments that it really has no performance impact. But you still do it. For the other reason.

The other reason has to do with fiduciary responsibility. If your bank or broker goes belly-up overnight, your cash holdings are likely gone. Sure, segregated accounts and so on, but in reality your cash is most likely going poof.

Your securities however are very likely to be returned to you, sooner or later. If you're dealing with a fraud situation, all bets are off, but if it's a regular case of greed, irresponsibility and some run-of-the-mill corporate corruption, you will likely see your securities returned at some point.

I'm going to keep the simulations simple here, and disregard this part. When deploying futures models in reality, you really do want to look closer at cash management. Both for fiduciary reasons, and for economic reasons.

Properly accounting for the historical impact of cash management, the simulation results would be a little bit higher. Some years, it made a substantial difference, when risk free returns were high.

Trend Model Source Code

The general structure of a Zipline simulation should be familiar to you by now, unless of course you skipped over the equity part of this book. As you will see in this code, the major difference with futures comes down to dealing with the dual concept of continuations versus contracts.

A continuation is a calculated time series, based on time series from individual contracts. Contract after contract are stitched together to form a long term time series, aiming to be as close as possible to the actual price impact of long term holding of the futures

market. As we don't have real long term series in the futures space, this is the best we can do to make something that we can run longer term time series analytics on. But you can't trade a continuation. It's just a calculation.

For trading, we need to figure out which contract is the relevant one to trade at the moment. That usually means the most liquid contract. What we do here is to construct a continuation which always uses the most traded contract as the current, and then when trading we ask the continuation to tell us which exact contract it's using at the moment. That's the one we will trade.

As futures contracts have a limited life span, this also means that we need to check every day if any positions need rolling.

The trading logic is not overly complicated. We start off by calculating the trend filter and the volatility measurement. Then we go over each market, one by one. If we already have an open position in a market, we check if

either of two possible stop conditions are met. Either if the price moved against us by three times the standard deviation, or if the trend direction flipped.

If there is no position held, we check if there is a new 50 day extreme in the direction of the trend, and if so we trade to open.

As before, I'll show you the key parts of the code bit by bit, with explanations, and in the end of this section you'll get the full source all at once.

At the top, we have various import statements as usual. Much of this should be familiar if you read the equity chapter and studied that code. This time however, there are a few futures specific functionality imported.

```
%matplotlib inline

import zipline
from zipline.api import future_symbol, \
    set_commission, set_slippage, schedule_function,
date_rules, \
    time_rules, continuous_future, order_target
from datetime import datetime
import pytz
```

```
import matplotlib.pyplot as plt
import pyfolio as pf
import pandas as pd
import numpy as np
from zipline.finance.commission import PerTrade,
PerContract
from zipline.finance.slippage import
VolumeShareSlippage, \
    FixedSlippage, VolatilityVolumeShare
```

In the equity momentum model, we were outputting a lot of text as the model was running, mostly to have something to look at for the minutes it takes to run the backtest. This doesn't affect anything apart from elevating boredom for those of us with short attention span.

In this model, I thought I'd show you another way of doing the same. Following this book's theme of incremental increase in complexity, this time I will show you how to update the text output, without printing a new row each time.

The equity momentum model prints one row per month, resulting in quite a few text lines to scroll down in the end. The approach here will update the same text, dynamically.

First we need to set it up, by importing the required libraries and creating an output variable which we can update during the backtest run.

```
# These lines are for the dynamic text reporting
from IPython.display import display
import ipywidgets as widgets
out = widgets.HTML()
display(out)
```

Now that this is set up, we can dynamically change the output text by setting out.valu e , which we'll do in a separate function as the backtest runs. The function below will be called every day.

We've seen before that the Zipline object called contex t can store just about anything you like for you during the backtest run. In this case, we'll use it to keep track of how many month's we've been trading so far. For this, all we need to do is to set context.months = 0 in initializ e . In the same startup routine, we'll schedule a monthly output report.

```
# We'll just use this for the progress output
# during the backtest. Doesn't impact anything.
context.months = 0
```

```
    # Schedule monthly report output
    schedule_function(
        func=report_result,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
```

The reporting routine itself only has a few lines of code. We update the number of months traded, calculate the annualized return so far, and update the text. That's all that's required to have a dynamic text update while we run the backtest.

```
def report_result(context, data):
    context.months += 1
    today = zipline.api.get_datetime().date()
    # Calculate annualized return so far
    ann_ret =
np.power(context.portfolio.portfolio_value /
starting_portfolio,
                 12 / context.months) - 1
```

```
    # Update the text
    out.value = """{} We have traded <b>{}</b>
months
    and the annualized return is <b>{:.2%}</
b>""".format(today, context.months, ann_ret)
```

So far, that's all just import statements and a bit of reporting, but we're getting to the actual model logic. As with the previous models, we have the

initializ e routine, where we set up commission, slippage and various other settings. With the equity model earlier, we needed a dynamic investment universe to reflect the stocks that would realistically have been on your radar in the past. With futures, it gets easier.

Here we just define the markets that we want to trade. It's a valid assumption that you would have picked more or less the same markets ten years ago.

In the equity model in chapter 12, we used a list of **symbol** objects as our investment universe. In Zipline logic, the **symbol** object applies only to stocks. For futures, we instead have two related concepts. We have **future_symbol** and **continuous_future** . The former refers to a specific futures contract, while the second refers to a calculated, long term price continuation based on many individual contracts.

As you can see in the initialize below, what we do here is to make a list of **continuous_future** objects, one for

each market.

```
def initialize(context):

    """
    Cost Settings
    """
    if enable_commission:
        comm_model = PerContract(cost=0.85,
exchange_fee=1.5)
    else:
        comm_model = PerTrade(cost=0.0)

    set_commission(us_futures=comm_model)

    if enable_slippage:

slippage_model=VolatilityVolumeShare(volume_limit=0.2)
    else:
        slippage_model=FixedSlippage(spread=0.0)

    set_slippage(us_futures=slippage_model)

    """
    Markets to trade
    """
    currencies = [
        'AD',
        'BP',
        'CD',
        'CU',
        'DX',
```

```
    'JY',
    'NE',
    'SF',
]
```

```
agricultural = [
    'BL',
    '_C',
    'CT',
    'FC',
    'KC',
    'LR',
    'LS',
    '_O',
    '_S',
    'SB',
    'SM',
    '_W',
]
nonagricultural = [
    'CL',
    'GC',
    'HG',
    'HO',
    'LG',
    'NG',
    'PA',
    'PL',
    'RB',
    'SI',
]
equities = [
    'ES',
    'NK',
```

```
        'NQ',
        'TW',
        'VX',
        'YM',
    ]
    rates = [
        'ED',
        'FV',
        'TU',
        'TY',
        'US',
    ]
```

```
    # List of all the markets
    markets = currencies + agricultural +
nonagricultural + equities + rates
```

```
    # Make a list of all continuations
    context.universe = [
        continuous_future(market, offset=0,
roll='volume', adjustment='mul')
            for market in markets
    ]
```

```
    # We'll use these to keep track of best position
reading
    # Used to calculate stop points.
    context.highest_in_position = {market: 0 for market
in markets}
    context.lowest_in_position = {market: 0 for market
in markets}
```

```

```

```
    # Schedule the daily trading
    schedule_function(daily_trade,
date_rules.every_day(), time_rules.market_close())
```

```
    # We'll just use this for the progress output
    # during the backtest. Doesn't impact anything.
    context.months = 0
```

```
    # Schedule monthly report output
    schedule_function(
        func=report_result,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )
```

We have a few helper functions here, just as we've seen in the equity chapter. There is the logic for checking if any contract needs to be rolled, which was discussed in the previous chapter. The logic for position size was also just explained in the section above on position allocation.

Now, what you're probably more eager to see is the code for the daily trading logic. We start off the daily_trad e routine by getting about a year's worth of continuous data for all the markets.

```
    # Get continuation data
    hist = data.history(
```

```
        context.universe,
        fields = ['close','volume'],
        frequency = '1d',
        bar_count = 250,
    )
```

Next we'll calculate the trend, and mentioned earlier we're calling bull if the 40 day EMA is higher than the 80 day EMA, as defined in our settings.

```
    # Calculate trend
    hist['trend'] =
hist['close'].ewm(span = fast_ma).mean() >
hist['close'].ewm(span = slow_ma).mean()
```

After we have the data and the trend info, we can iterate each market, one by one. When we iterate market by market, we first check if there is a position on or not. If a position is held, we check if it's long or short and run the applicable logic. If a position is not held, we check if there is a bull market or a bear market at the moment, and run the relevant code to see if a position should be opened or not.

Here is it, bit by bit. First we start the loop and prepare the data we need, including a standard deviation calculation.

```
    # Iterate markets, check for trades
    for continuation in context.universe:
```

```
        # Get root symbol of continuation
        root = continuation.root_symbol
```

```
        # Slice off history for just this market
        h = hist.xs(continuation, 2)
```

```
        # Get standard deviation
        std = h.close.diff()[-vola_window:].std()
```

Then we process long positions.

```
        if root in open_pos: # Position is open

            # Get position
            p =
context.portfolio.positions[open_pos[root]]
```

```
            if p.amount > 0: # Position is long
                if context.highest_in_position[root] == 0:
# First day holding the position
                    context.highest_in_position[root] =
p.cost_basis
                else:
                    context.highest_in_position[root] = max(
                        h['close'].iloc[-1],
context.highest_in_position[root]
                    )
```

```
                # Calculate stop point
                stop = context.highest_in_position[root] -
```

```
(std  * stop_distance)
            # Check if stop is hit
            if h.iloc[-1]['close'] < stop:
                contract = open_pos[root]
                order_target(contract, 0)
                context.highest_in_position[root] = 0
            # Check if trend has flipped
            elif h['trend'].iloc[-1] == False:
                contract = open_pos[root]
                order_target(contract, 0)
                context.highest_in_position[root] = 0
```

Process short positions.

```
        else: # Position is short
            if context.lowest_in_position[root] == 0: #
First day holding the position
                context.lowest_in_position[root] =
p.cost_basis
            else:
                context.lowest_in_position[root] = min(
                    h['close'].iloc[-1],
context.lowest_in_position[root]
                )
```

```
            # Calculate stop point
            stop = context.lowest_in_position[root] +
(std  * stop_distance)
```

```
            # Check if stop is hit
            if h.iloc[-1]['close'] > stop:
                contract = open_pos[root]
                order_target(contract, 0)
                context.lowest_in_position[root] = 0
```

```
            # Check if trend has flipped
            elif h['trend'].iloc[-1] == True:
                contract = open_pos[root]
                order_target(contract, 0)
                context.lowest_in_position[root] = 0
```

If no position is on, we deal with bull market scenario.

```
        else: # No position on
            if h['trend'].iloc[-1]: # Bull trend
                # Check if we just made a new high
                if h['close'][-1] == h[-breakout_window:]
['close'].max():
                    contract = data.current(continuation,
'contract')

                    contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                                std, \

contract.price_multiplier)
```

```
                    # Limit size to 20% of avg. daily volume
                    contracts_cap = int(h['volume']
[-20:].mean() * 0.2)
                    contracts_to_trade =
min(contracts_to_trade, contracts_cap)
```

```
                    # Place the order
                    order_target(contract, contracts_to_trade)
```

And finally, we deal with bear

markets.

```
        else: # Bear trend
            # Check if we just made a new low
            if h['close'][-1] == h[-breakout_window:]
['close'].min():
                contract = data.current(continuation,
'contract')

                contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                                    std, \

contract.price_multiplier)
```

```
                # Limit size to 20% of avg. daily volume
                contracts_cap = int(h['volume']
[-20:].mean() * 0.2)
                contracts_to_trade =
min(contracts_to_trade, contracts_cap)
```

```
                # Place the order
                order_target(contract, -1 *
contracts_to_trade)
```

Finally, still in the daily_trad e routine, if there are positions open, we execute the function to check if anything needs to be rolled. This was explained more in detail in chapter 14.

```
    # If we have open positions, check for rolls
```

```
    if len(open_pos) > 0:
        roll_futures(context, data)
```

As your Python skills should be continuingly improving throughout this book, I will increasingly rely on comments in the code and not explain every line with text. If I did, this book would go from around 500 pages to the double. I focus instead on explaining new concepts and important segments.

Now that we have looked at the individual parts of the code, you may want to have a look at the complete model code below. As with all other code in this book, you can download it from the book website, **www.followingthetrend.com/trading-evolved** .

```
%matplotlib inline

import zipline
from zipline.api import future_symbol, \
    set_commission, set_slippage, schedule_function,
date_rules, \
    time_rules, continuous_future, order_target
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import pyfolio as pf
```

```
import pandas as pd
import numpy as np
from zipline.finance.commission import PerTrade,
PerContract
from zipline.finance.slippage import
VolumeShareSlippage, \
    FixedSlippage, VolatilityVolumeShare

# These lines are for the dynamic text reporting
from IPython.display import display
import ipywidgets as widgets
out = widgets.HTML()
display(out)

"""
Model Settings
"""
starting_portfolio = 50000000
risk_factor = 0.0015
stop_distance = 3
breakout_window = 50
vola_window = 40
slow_ma = 80
fast_ma = 40
enable_commission = True
enable_slippage = True


def report_result(context, data):
    context.months += 1
    today = zipline.api.get_datetime().date()
    # Calculate annualized return so far
    ann_ret =
np.power(context.portfolio.portfolio_value /
```

```
starting_portfolio,
                12 / context.months) - 1
```

```
    # Update the text
    out.value = """{} We have traded <b>{}</b>
months
    and the annualized return is <b>{:.2%}</
b>""".format(today, context.months, ann_ret)

def roll_futures(context, data):
    open_orders = zipline.api.get_open_orders()
```

```
    for held_contract in context.portfolio.positions:
        # don't roll positions that are set to change by
core logic
        if held_contract in open_orders:
            continue
```

```
        # Save some time by only checking rolls for
        # contracts stopping trading in the next days
        days_to_auto_close = (
            held_contract.auto_close_date.date() -
data.current_session.date()
        ).days
        if days_to_auto_close > 5:
            continue
```

```
        # Make a continuation
        continuation = continuous_future(
                held_contract.root_symbol,
                offset=0,
                roll='volume',
```

```
            adjustment='mul'
            )
```

```
        # Get the current contract of the continuation
        continuation_contract =
data.current(continuation, 'contract')
```

```
        if continuation_contract != held_contract:
            # Check how many contracts we hold
            pos_size =
context.portfolio.positions[held_contract].amount
            # Close current position
            order_target(held_contract, 0)
            # Open new position
            order_target(continuation_contract, pos_size)
```

```
def position_size(portfolio_value, std, point_value):
    target_variation = portfolio_value * risk_factor
    contract_variation = std * point_value
    contracts = target_variation / contract_variation
    return int(np.nan_to_num(contracts))
```

```
def initialize(context):
```

```
    """
    Cost Settings
    """
    if enable_commission:
        comm_model = PerContract(cost=0.85,
exchange_fee=1.5)
    else:
        comm_model = PerTrade(cost=0.0)
```

```
    set_commission(us_futures = comm_model)

    if enable_slippage:

slippage_model = VolatilityVolumeShare(volume_limit = 0.2)
    else:
        slippage_model = FixedSlippage(spread = 0.0)

    set_slippage(us_futures = slippage_model)

    """
    Markets to trade
    """
    currencies = [
        'AD',
        'BP',
        'CD',
        'CU',
        'DX',
        'JY',
        'NE',
        'SF',
    ]

    agricultural = [
        'BL',
        '_C',
        'CT',
        'FC',
        'KC',
        'LR',
```

```
    'LS',
    '_O',
    '_S',
    'SB',
    'SM',
    '_W',
]
nonagricultural = [
    'CL',
    'GC',
    'HG',
    'HO',
    'LG',
    'NG',
    'PA',
    'PL',
    'RB',
    'SI',
]
equities = [
    'ES',
    'NK',
    'NQ',
    'TW',
    'VX',
    'YM',
]
rates = [
    'ED',
    'FV',
    'TU',
    'TY',
    'US',
]
```

```
    # Make a list of all the markets
    markets = currencies + agricultural +
nonagricultural + equities + rates
```

```
    # Make a list of all continuations
    context.universe = [
        continuous_future(market, offset=0,
roll='volume', adjustment='mul')
            for market in markets
    ]
```

```
    # We'll use these to keep track of best position
reading
    # Used to calculate stop points.
    context.highest_in_position = {market: 0 for market
in markets}
    context.lowest_in_position = {market: 0 for market
in markets}
```

```
    # Schedule the daily trading
    schedule_function(daily_trade,
date_rules.every_day(), time_rules.market_close())
```

```
    # We'll just use this for the progress output
    # during the backtest. Doesn't impact anything.
    context.months = 0
```

```
    # Schedule monthly report output
    schedule_function(
        func=report_result,
```

```
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )
```

```
def analyze(context, perf):
    returns, positions, transactions =
pf.utils.extract_rets_pos_txn_from_zipline(perf)
    pf.create_returns_tear_sheet(returns,
benchmark_rets=None)
```

```
def daily_trade(context, data):
    # Get continuation data
    hist = data.history(
        context.universe,
        fields=['close','volume'],
        frequency='1d',
        bar_count=250,
    )
```

```
    # Calculate trend
    hist['trend'] =
hist['close'].ewm(span=fast_ma).mean() >
hist['close'].ewm(span=slow_ma).mean()
```

```
    # Make dictionary of open positions
    open_pos = {
        pos.root_symbol: pos
        for pos in context.portfolio.positions
    }
```

```
    # Iterate markets, check for trades
    for continuation in context.universe:
```

```
        # Get root symbol of continuation
        root = continuation.root_symbol

        # Slice off history for just this market
        h = hist.xs(continuation, 2)

        # Get standard deviation
        std = h.close.diff()[-vola_window:].std()

        if root in open_pos: # Position is open

            # Get position
            p =
context.portfolio.positions[open_pos[root]]

            if p.amount > 0: # Position is long
                if context.highest_in_position[root] == 0:
# First day holding the position
                    context.highest_in_position[root] =
p.cost_basis
                else:
                    context.highest_in_position[root] = max(
                        h['close'].iloc[-1],
context.highest_in_position[root]
                    )

                # Calculate stop point
                stop = context.highest_in_position[root] -
(std  * stop_distance)
                # Check if stop is hit
                if h.iloc[-1]['close'] < stop:
```

```
                contract = open_pos[root]
                order_target(contract, 0)
                context.highest_in_position[root] = 0
            # Check if trend has flipped
            elif h['trend'].iloc[-1] == False:
                contract = open_pos[root]
                order_target(contract, 0)
                context.highest_in_position[root] = 0
```

```
        else: # Position is short
            if context.lowest_in_position[root] == 0: #
First day holding the position
                context.lowest_in_position[root] =
p.cost_basis
            else:
                context.lowest_in_position[root] = min(
                    h['close'].iloc[-1],
context.lowest_in_position[root]
                )
```

```
            # Calculate stop point
            stop = context.lowest_in_position[root] +
(std  * stop_distance)
```

```
            # Check if stop is hit
            if h.iloc[-1]['close'] > stop:
                contract = open_pos[root]
                order_target(contract, 0)
                context.lowest_in_position[root] = 0
            # Check if trend has flipped
            elif h['trend'].iloc[-1] == True:
                contract = open_pos[root]
                order_target(contract, 0)
```

```
                context.lowest_in_position[root] =
0
```

```
        else: # No position on
            if h['trend'].iloc[-1]: # Bull trend
                # Check if we just made a new high
                if h['close'][-1] == h[-breakout_window:]
['close'].max():
                    contract = data.current(continuation,
'contract')

                    contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                                std, \

contract.price_multiplier)
```

```
                    # Limit size to 20% of avg. daily volume
                    contracts_cap = int(h['volume']
[-20:].mean() * 0.2)
                    contracts_to_trade =
min(contracts_to_trade, contracts_cap)
```

```
                    # Place the order
                    order_target(contract, contracts_to_trade)
```

```
            else: # Bear trend
                # Check if we just made a new low
                if h['close'][-1] == h[-breakout_window:]
['close'].min():
                    contract = data.current(continuation,
```

```
'contract')

                contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                        std, \

contract.price_multiplier)
```

```
                # Limit size to 20% of avg. daily volume
                contracts_cap = int(h['volume']
[-20:].mean() * 0.2)
                contracts_to_trade =
min(contracts_to_trade, contracts_cap)
```

```
                # Place the order
                order_target(contract, -1 *
contracts_to_trade)
```

```
    # If we have open positions, check for rolls
    if len(open_pos) > 0:
        roll_futures(context, data)
```

```
start = datetime(2001, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2019, 1, 31, 8, 15, 12, 0, pytz.UTC)

perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    analyze=analyze,
    capital_base=starting_portfolio,
    data_frequency = 'daily',
```

```
bundle = 'futures' )
```

Core Trend Model Results

After running the model, the first thing to do is to check the general shape of the equity curve. That's of course not a valid or measurable basis for decisions, but it does give a general overview of whether or not a model is worth investigating any further. You will find that you can often discard ideas and concepts after a glance at the equity curve. Sometimes it will give you much more information about what a strategy will do than the usual ratios, such as Sharpe, annualized return or maximum drawdowns.

Table 15.1 - Core Trend Monthly Returns

Year	[illegible]
20	+3.1
2002	+[illegible]
20	+17.4
2004	+[illegible]

20	-1.2
2006	+14.8
20	+18.8
2008	+29.6
20	-0.7
2010	+20.8
20	-5.0
2012	+16.8
20	+0.5
2014	+23.8
20	+6.2
2016	+19.8
20	+1.3
2018	+28.8

Figure 15-2 shows the backtest performance of this simple trend model, compared to that of the S&P 500 Total Return Index. This basic visual overview gives a few important pieces of information. The most obvious being that it seems as if this strategy has a higher long term performance than the S&P 500. But that's almost irrelevant. You could even argue that it's an irrelevant comparison to begin with. After all, this is not an equity related strategy in any way.

Simply having a higher long term

performance is not a measurement of quality. Keep in mind that in a simulation like this, we could simply scale position risk sizes up or down to change the end cumulative performance numbers. Comparing the start and end points only does not make much sense. We need to consider how we got there.

Next we can see in the same figure how the performance tends to come at different times than the stock markets. There were two major equity bear markets during this period and both times the futures strategy did well. This should really not be a surprise to anyone who has been following the trend following space. The approach has historically done very well during periods of market crisis.

This also means that the trend model at times severely underperforms during bull markets, and that can be a bigger problem than one might think. The stock market is used as a benchmark in this figure, even though the strategy really has very little to do with that sector. The point is that the public at large and even financial professionals tend to compare

everything with the stock markets.

If you manage other people's money, this is something you will quickly notice. Even if you have a strategy which really should be unrelated to the stock markets, you will still always be compared to it. If you lose ten percent and the stock market loses 15 percent, no one will complain. If you gain ten percent and the stock market gains 15, you may find that some clients are unhappy. Like it or not, the stock market is the de facto benchmark.

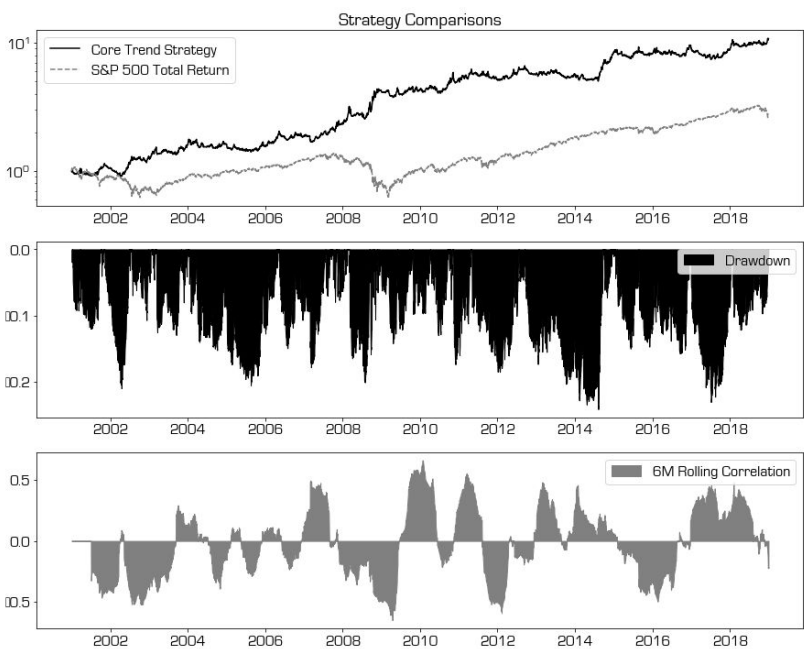


Figure 15-2 - Core Trend Model Equity

Curve

While having a low correlation to equity markets during bull markets can be problematic, having a low correlation during bear markets is of vital importance. This is where a strategy like this really shines. People like making money, but there is nothing like making money while your neighbor loses his.

The return chart also shows a somewhat worrying aspect. A quick glance tells you that returns seem to be going down. We are seeing deeper and longer lasting drawdowns. Yes, this is true. And there are good reasons for this. The environment for trend following has not been optimal in the past few years.

The low volatility situation is one issue. Trend following thrives on high volatility markets and has performed remarkably well during volatile bear markets. But we have seen a ten year bull market with fairly low volatility, and that has reduced earning potential for diversified futures strategies.

The low interest environment has

also been a concern. Historically, trend following made outsized returns on simply being long bonds, treasuries and money market futures, capitalizing on the slowly decreasing yields. As yields came down to historically low levels and stagnated, that source of income dwindled.

It would however be a mistake to take the last few years as a sign of trend following demise. Just like it was a mistake to take the extreme returns of 2008 and extrapolate that into the future. What we do know is that this highly simplified type of strategy has worked well for decades. It has outperformed traditional investment approaches and it has shown very strong performance during bear markets and times of market distress. It would be a reasonable assumption that it will perform well during the next bear market.

Table 15.2 Holding Period Returns Core Trend Model

[illegible]
[illegible]
[illegible]

of 9% during this period.

While this graph tells you little about important details, such as the volatility and drawdowns it took to get those returns, it can help provide a long term perspective.

16 Time Return Trend Model

The time return trend model is probably the simplest trend model you will ever see. It may be the simplest trading model of any kind that you will encounter. At least if you only count reasonably well performing models. There is nothing wrong with the results from this model. It's not the most practical model to trade, but the returns have been nothing short of stellar, given how simple the rules really are. And of course, I'm about to give you all the source code.

The aim of this model is to capture long term trends while avoiding common

problem of early stop out. All too often, classic trend following models suffer from taking their stops too early on short term pullbacks, only to see prices return back to the trend again. While these classic trend models tend to show strong long term performance despite this issue, the time return trend model is not susceptible to the problem at all.

The rules are so simple that it's easy to dismiss this model at first glance. I would urge you to take it seriously though. There is plenty to be learned from understanding this model.

You should bear in mind from the start that this particular model is not very practical to trade. For most readers, it simply is not possible to trade it. To actually implement this model, you will need a substantial sum of money, a large number of positions would be open at any given time and you would constantly run on a very high margin to equity ratio.

But, this model remains a very valuable learning tool. If you understand the mechanics, you can adapt the model

and make it more practical. This version that I will show you is for learning, not for you to go and implement.

Investment Universe

Just as with the previous model, we will use a very broad investment universe. A model like this is completely dependent on diversification and will not perform well on a small investment universe. If you try rules like this on any given market, or a small number of markets, you'll likely see results ranging from poor to mediocre. The effect comes from trading a broad set of markets, all at once.

For this model, we're going to trade the same set of broad futures markets that we used in the previous model. We're covering equity indexes, commodities, currencies, treasuries and money markets.

Trading Frequency

With this particular model, we only trade once per month. We completely

ignore anything that happens during that month. No matter how large the moves, no action is taken during the month. No stop point, no profit targets. Just make your trades at the start of the month and go fishing for a while.

Actually, there is one thing that we need to do between the monthly trade points. As we are dealing with futures here, we do need to check for rolls daily. That is, we need to make sure that we are holding the correct contract, as liquidity shifts from one contract delivery to another. Just like for any futures model, this is something that we won't get away from.

Position Allocation

For this model, we will use the same position allocation logic as we employed for the previous model. We measure volatility using standard deviation, and set position sizes based on inverse volatility logic. That is, we have a lower face value amount of something

more volatile, and vice versa, in an attempt to target an approximate equal volatility, or risk if you will, per position.

To keep the logic simple here, no regular rebalancing of position sizes is done, nor is any more complex volatility targeting technique employed. Both of these make more sense for institutional portfolio management than for individual accounts.

Trading Rules

As already mentioned, this model trades only once per month. At that time we check only two things.

Is the price higher or lower than a year go?

Is the price higher or lower than half a year ago?

That's all we care about here. For this model, there are no breakout indicators, no moving averages, and no indicators of any kind. It is, quite purposely, an extremely simple trend

model.

Note that we do the calculations here based on the continuations, as we did in the previous model, not on individual contracts. That's generally how we need work with futures models, as the individual contracts have such limited time series history.

Dynamic Performance Chart

Since this model is quite simple, this seems like a good place to throw in a new neat little trick to teach you. If you have replicated the models and run the backtests in the previous chapters, you will have noticed by now that it sometimes takes a little while to finish them. I showed you earlier simple ways to output some text, such as last month's performance during the run. But this is neither pretty, nor very informative.

Wouldn't it be nice if we could get a dynamically updating graph while we

wait? A graph showing the ongoing performance of the simulation, as it happens. It's a well-known trick that people are less annoyed with waiting times if there is something to look at or listen to. That's after all why they have mirrors in elevators.

As it turns out, it's not only possible but even quite simple to have a dynamic chart display as you run the backtest. From now on in this book, I will use this way to output results as the backtests are under way.

To set up these dynamic charts in the **Jupyter Notebook** environment, we need to do three simple things.

First, until now we have been using the line `%matplotlib inlin e` at the top of our code. That tells the charting library, **matplotlib** , to show the graphs in the output as images when they are requested by our code. But using that does not allow for interactive charts.

To be able to get interactive charts, we need to change that line to `%matplotlib noteboo k` .

Second, we need to add some brief code to make a **DataFrame** for storing the charting data, as well as initializing the chart.

```
# DataFame for storing and updating the data that we want to graph
dynamic_results = pd.DataFrame()
# Init figure
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111)
ax.set_title('Time Return Performance')
```

Now the only thing left is to update the data as we move along in the backtest. To keep it clean, I added a separate scheduled function in this example. Keep in mind that if you are looking for pure speed, using a dynamic graph will slow you down, and the more so the more often you update it.

I added this row to our initializ e function, which will update the graph once a month. If you have really short attention span you could update this daily, but performance will suffer.

```
schedule_function(update_chart,
date_rules.month_start(), time_rules.market_close())
```

And then this new function to store

and update the graph.

```
def update_chart(context,data):
    # This function continuously update the graph
during the backtest
    today = data.current_session.date()
    dynamic_results.loc[today, 'PortfolioValue'] =
context.portfolio.portfolio_value
```

```
    if ax.lines: # Update existing line
        ax.lines[0].set_xdata(dynamic_results.index)

ax.lines[0].set_ydata(dynamic_results.PortfolioValue)
    else: # Create new line
        ax.semilogy(dynamic_results)
```

```
    # Update scales min/max
    ax.set_ylim(
        dynamic_results.PortfolioValue.min(),
        dynamic_results.PortfolioValue.max()
    )
    ax.set_xlim(
        dynamic_results.index.min(),
        dynamic_results.index.max()
    )
```

```
    # Redraw the graph
    fig.canvas.draw()
```

Time Return Source

Code

The code for this model is really quite simple. Even if you don't really know anything about programming, you can still figure it out. Compared to most programming languages, Python is quite easy to read and understand.

As usual, the very top you will find some import statements, telling our code which libraries we intend to use. Then there are a few parameters defined. Feel free to play around with these. You can change the risk factor, the volatility calculation window, liquidity filter, and trend windows.

You will notice that there are two settings for trend window. One short and one long. One is set to 125 days and the other 250, or approximately half a year and a full year respectively. The rules, as you see, requires positive return over both time periods for a long position, or negative over both for a short position.

In the Initializ e function we can enable or disable cost and slippage. Again,

test it out and see how it changes things. It wouldn't make much sense for me to simply tell you. Trust me, you will learn much more if you get this code up and running locally, and see for yourself.

Then, in the same function, the investment universe is defined before we set schedulers to run the rebalancing monthly and the futures roll check daily.

In the monthly rebalancing, we first check if a position is already open or not for each market. If there is no position on, we check if yesterday's price is higher than both a year ago and half a year ago. If so, buy. If it's lower than both those two points in time, go short. And else, hold no position at all.

```
%matplotlib notebook

import zipline
from zipline.api import future_symbol,  \
    set_commission, set_slippage, schedule_function,
date_rules, \
    time_rules, continuous_future, order_target
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import matplotlib
import pyfolio as pf
```

```
import pandas as pd
import numpy as np
from zipline.finance.commission import PerShare,
PerTrade, PerContract
from zipline.finance.slippage import
VolumeShareSlippage, \
    FixedSlippage, VolatilityVolumeShare

"""
Model Settings
"""
starting_portfolio = 10000000
risk_factor = 0.0015
vola_window = 60
short_trend_window = 125
long_trend_window = 250

"""
Prepare for dynamic chart
"""
dynamic_results = pd.DataFrame()
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111)
ax.set_title('Time Return Performance')


def initialize(context):
    """
    Cost Settings
    """
    context.enable_commission = True
    context.enable_slippage = True


    if context.enable_commission:
```

```
        comm_model = PerContract(cost=0.85,
exchange_fee=1.5)
    else:
        comm_model = PerTrade(cost=0.0)
```

```
    set_commission(us_futures=comm_model)
```

```
    if context.enable_slippage:

slippage_model=VolatilityVolumeShare(volume_limit=0.2)
    else:
        slippage_model=FixedSlippage(spread=0.0)
```

```
    set_slippage(us_futures=slippage_model)
```

```
    currencies = [
        'AD',
        'BP',
        'CD',
        'CU',
        'DX',
        'JY',
        'NE',
        'SF',
    ]
    agriculturals = [
        'BL',
        'BO',
        '_C',
        'CC',
        'CT',
        'FC',
```

```
    'KC',
    'LB',
    'LC',
    'LR',
    'LS',
    '_O',
    '_S',
    'SB',
    '_W',
]
nonagriculturals = [
    'CL',
    'GC',
    'HG',
    'HO',
    'LG',
    'NG',
    'PA',
    'PL',
    'RB',
    'SI',
]
equities = [
    'ES',
    'NK',
    'NQ',
    'TW',
    'VX',
    'YM',
]
rates = [
    'ED',
    'FV',
    'TU',
```

```
        'TY',
        'US',
    ]
```

```
    # Join sector lists into one list
    markets = currencies + agriculturals +
nonagriculturals + equities + rates
```

```
    # Make a list of all continuations
    context.universe = [
        continuous_future(market, offset=0,
roll='volume', adjustment='mul')
            for market in markets
    ]
```

```
    # Schedule daily trading
    schedule_function(rebalance,
date_rules.month_start(), time_rules.market_close())
```

```
    # Schedule daily roll check

schedule_function(roll_futures,date_rules.every_day(),
time_rules.market_close())
```

```
    # Schedule monthly chart update

schedule_function(update_chart,date_rules.month_start(),
time_rules.market_close())
```

```
def update_chart(context,data):
    # This function continuously update the graph
```

```
during the backtest
    today = data.current_session.date()
    dynamic_results.loc[today, 'PortfolioValue'] =
context.portfolio.portfolio_value
```

```
    if ax.lines: # Update existing line
        ax.lines[0].set_xdata(dynamic_results.index)

ax.lines[0].set_ydata(dynamic_results.PortfolioValue)
    else: # Create new line
        ax.semilogy(dynamic_results)
```

```
    # Update scales min/max
    ax.set_ylim(
        dynamic_results.PortfolioValue.min(),
        dynamic_results.PortfolioValue.max()
    )
    ax.set_xlim(
        dynamic_results.index.min(),
        dynamic_results.index.max()
    )
```

```
    # Redraw the graph
    fig.canvas.draw()

def roll_futures(context,data):
    today = data.current_session.date()
    open_orders = zipline.api.get_open_orders()
    for held_contract in context.portfolio.positions:
        if held_contract in open_orders:
            continue
        days_to_auto_close =
(held_contract.auto_close_date.date() - today).days
```

```
        if days_to_auto_close > 10:
            continue
```

```
        # Make a continuation
        continuation = continuous_future(
                held_contract.root_symbol,
                offset=0,
                roll='volume',
                adjustment='mul'
                )
        continuation_contract =
data.current(continuation, 'contract')
```

```
        if continuation_contract != held_contract:
            pos_size =
context.portfolio.positions[held_contract].amount
            order_target(held_contract, 0)
            order_target(continuation_contract, pos_size)

def position_size(portfolio_value, std, pv, avg_volume):
    target_variation = portfolio_value * risk_factor
    contract_variation = std * pv
    contracts = target_variation / contract_variation
    return int(np.nan_to_num(contracts))

def rebalance(context, data):
    # Get the history
    hist = data.history(
        context.universe,
        fields=['close', 'volume'],
        frequency='1d',
        bar_count=long_trend_window,
    )
```

```
    # Make a dictionary of open positions
    open_pos = {pos.root_symbol: pos for pos in
context.portfolio.positions}

    # Loop all markets
    for continuation in context.universe:
        # Slice off history for this market
        h = hist.xs(continuation, 2)
        root = continuation.root_symbol

        # Calculate volatility
        std = h.close.diff()[-vola_window:].std()

        if root in open_pos: # Position is already open
            p =
context.portfolio.positions[open_pos[root]]
            if p.amount > 0: # Long position
                if h.close[-1] < h.close[-
long_trend_window]:
                    # Lost slow trend, close position
                    order_target(open_pos[root], 0)
                elif h.close[-1] < h.close[-
short_trend_window]:
                    # Lost fast trend, close position
                    order_target(open_pos[root], 0)
            else: # Short position
                if h.close[-1] > h.close[-
long_trend_window]:
                    # Lost slow trend, close position
                    order_target(open_pos[root], 0)
```

```
            elif h.close[-1] > h.close[-
short_trend_window]:
                # Lost fast trend, close position
                order_target(open_pos[root], 0)
```

```
        else:   # No position open yet.
            if (h.close[-1] > h.close[-long_trend_window])
\
                and \
                (h.close[-1] > h.close[-
short_trend_window]):
                    # Buy new position
                    contract = data.current(continuation,
'contract')
                    contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                        std, \
                                        contract.price_multiplier, \
                                        h['volume'][-20:].mean())
```

```
                    order_target(contract, contracts_to_trade)
            elif (h.close[-1] < h.close[-
long_trend_window]) \
                and \
                (h.close[-1] < h.close[-
short_trend_window]):
                    # New short position
                    contract = data.current(continuation,
'contract')
                    contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
```

```
                                    std, \
                                    contract.price_multiplier, \
                                    h['volume'][-20:].mean())

                order_target(contract, contracts_to_trade
*-1)

start = datetime(2001, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2018, 12, 31, 8, 15, 12, 0, pytz.UTC)

perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    capital_base=starting_portfolio,
    data_frequency = 'daily',
    bundle='futures' )
```

Time Return Model Performance

Our model is so incredibly simple that it couldn't possibly show any sort of interesting performance. Right? Well, as it turns out, trend following does not have to be complex. Sure, this model is still overly simplified and can easily be improved upon, but even in this simple state it still works.

A quick glance at the monthly

return table and the long term chart should tell us that we should at least not dismiss this kind of approach out of hand. We can see a few interesting features. First, it tends to perform well during bear markets. We only had two severe bear markets during this time span, but the model handled both really well. Second, we can see that the model appears to be showing quite strong long term performance, at acceptable drawdowns.

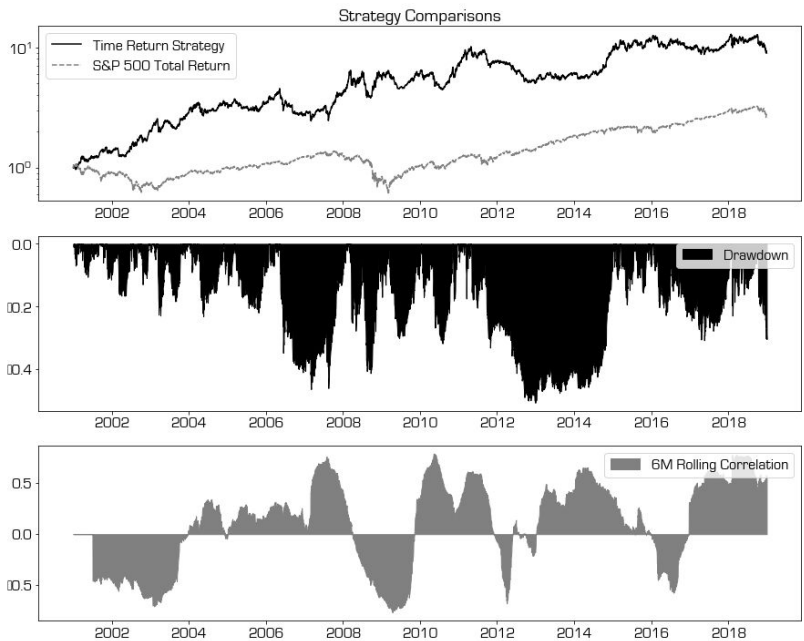


Figure 16-1 - Time Momentum Performance

Table 16.1 - Time Return, Monthly Figures

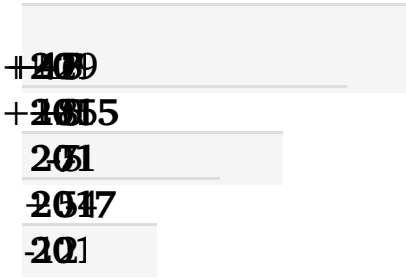
Year	[illegible]
2001	+[illegible]
2002	+[illegible]
20[illegible]	+36.0
2004	+[illegible]
20[illegible]	+1.1
2006	+[illegible]
20[illegible]	+49.4
2008	+[illegible]
20[illegible]	+4.6
2010	+[illegible]
20[illegible]	+0.3
2012	-[illegible]
20[illegible]	+10.3
2014	+[illegible]
20[illegible]	+18.0
2016	+[illegible]
20[illegible]	+5.2
2018	-[illegible]

What we have here is certainly not the best possible model in any kind, but it's not utterly crazy either. The volatility is quite high, and most likely far too high for most people's appetite. But this kind of performance with such extremely simple

rules should tell you something about the nature of trend following.

There are no indicators at all used here. No moving averages, no RSI, no Stochastics, no MACD, no technical analysis terms of any kind required. All we are doing is checking two price points. And we only do that once every month. Keep that in mind the next time someone offers to sell you some amazing trend following system.

Table 16.2 Holding Period Analysis



But there is one more observation that you should have made by now. Have we not seen this return profile before? Is this not a little similar to something we have seen earlier?

Yes, dear reader. And that's exactly the point that I'm somewhat slowly trying to make.

Trend following returns come from a simple concept. Most trend following models have very high correlation to each other. Or in other words, there are not a whole lot of different ways that you can follow trends. And that's why most trend following trading models have high correlation, and why most trend following hedge funds look very similar over time. The biggest differentiators are asset mix and time frame, and in this case I purposely kept both models more or less the same.

As seen in Figure 16-2, these two very different models have far from perfect correlation, but over time they tend to even out and deliver more or less the same result. You could easily make scores of seemingly different trend models, and they will probably all look very similar when compared on an equal basis.

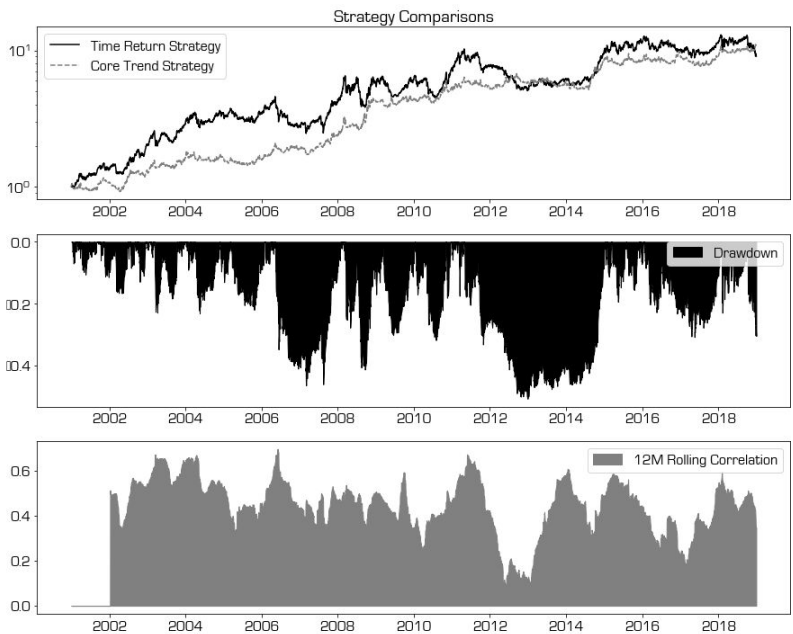


Figure 16-2 Comparing Trend models

The first and foremost conclusion we have to draw from this is that trend following, as a phenomenon is quite

simple. The bulk of trend following returns can be captured with rules that could be summarized in a single sentence.

As for the returns, if you just look at the yearly numbers they are remarkably strong. Since 2000, only three years ended in the red. No year lost more than 25%, while the best year doubled the money. If you had placed 100,000 in such a strategy in January 2000, you would have over two and a half million in 2016. But of course, no one gets to trade real money on historical simulations.

While this strategy may seem quite attractive when you look back at a long term simulation, it's far from a polished strategy. The volatility for this model is at times far above acceptable levels. The volatility always needs to be taken into account when analyzing a trading model, and in chapter 22 you will get a better insight into how to do this.

The drawdown in 2008 is also a bit worrying. What happened there was that first the model made large gains, and then it much of it was given back. Looking at a

simulation, that may not seem like a big deal. Easy come, easy go, right? Perhaps so, if you have the benefit of hindsight in picking your entry into the strategy. If you had invested into this strategy just after the gains, you would just have seen the losses and it would have taken three years to recover them. In reality, you never know when the drawdown will come. Assume it will happen at the worst possible time. Because that's how reality rolls.

Before going too deep into analyzing the returns of this simple 12 months return model, we should identify and fix a couple of deficiencies.

Rebalancing

This is a very long term trading model with an average holding period of about half a year. Such a model absolutely needs a rebalancing mechanism. Without one, the position risk, and in extension the portfolio risk, would eventually be quite random.

The position allocation for this model is based on recent volatility. Remember how we look at past volatility and calculate position sizes with the aim of allocating an equivalent amount of risk to each position. Optimally, we would like every position to have an equal possibility to impact the portfolio on a daily basis. That means buying more of slow moving markets and buying less of the more volatile markets.

But of course, the markets are not going to just politely maintain the same volatility level just to make your calculations easier. Volatility is not static and therefore your allocation can't be either.

The solution to this is to reset the position sizes on a regular basis. If you don't, you lose control over position risk and that will cause some nasty spikes in the equity curve.

There are two ways that the actual position risk can change over time. The volatility of the market can, and will change over time. If you would like to

build upon this model and improve it, the first order of business should probably be to look at rebalancing and maintaining risk targets.

17 Counter Trend Trading

Perhaps trend following is not your cup of tea. After all, it can be quite a frustrating type of strategy to trade at times. Even if results in the long run tends to turn out well, anyone who has done some serious trend following trading can tell you how it often feels like you could make money from doing the exact opposite.

One thing in particular that most trend followers see time and again is how prices fall back to the trailing stops, only to move right back into the trend soon after you are stopped out. Out of necessity, trend following models need to stop out at some point. It's something unavoidable for that type of strategy. You

may think that the solution is to simply move the trailing stop distance for the trend models further away, but it's not that easy. That will result in a longer term trend model, and you still have the same issue on a different time horizon.

This is quite an interesting and easily observable market phenomenon. As usual, we can only speculate as to why this happens and no matter how good of an explanation we come up with, we can never be sure if it's the real reason or not. Short of installing listening devices in the homes and workplaces of all the world's traders, we simply can't know why market participants react the way that they do, and it would appear as if Amazon Alexa is already way ahead of us in that game.

But, the inability to verify or even make practical use of an origin theory of a market phenomenon shouldn't prevent us from fabricating some. A successful trading model often has a good story behind it. Sometimes that story may help create confidence in the approach, and sometimes it may help raise the necessary assets to trade it. Often a solid trading

model starts off with such a theory, with the broad idea of what you want to take advantage of, and why. Surprising as it may seem, it can actually help if there is a logical reason for your trades.

In this case, there is a possible explanation that comes to mind. The trend following industry was once a niche trading variant which few took very seriously. As the success of this strategy became hard to ignore, it attracted increasing amounts of both new traders and investors. At this time, there is in excess of a quarter of a trillion dollars under management with such strategies.

It would be reasonable to assume that this kind of size has a serious market impact. Trend followers were once passive market followers. Now, one could argue, they are not merely following market price moves but creating them.

Contrary to some public perception, being large enough to move market prices is rarely an advantage. It's not like you can snap your fingers and make prices lower before you want to buy. The case is

more likely that you would like to buy, but as you start doing so, your order sizes push the prices higher and gives you a significantly worse execution price. It can be a real problem to have too much money in this business. Should David Harding by any chance be reading this book, I'm perfectly willing to be a good friend and help out by taking over some of those assets, Dave.

In my first book, I demonstrated that most trend following hedge funds employ very similar strategies. This would imply that they all have mechanisms similar to trailing stop losses. If this is true, that would mean that they all trade at very similar times.

If we had a nice bull market trend in the oil market for instance, it would reasonable to assume that most if not all trend followers are long. Now imagine that the price of oil starts to decline for whatever reason. At some point, trend followers start exit. And that can push the price a bit further down, perhaps even triggering stops for other trend followers.

Given the sheer size of trend following capital, this can create a situation where a small market correction is amplified by their stop loss orders. This could result in two things. First that the market is pushed further back than it otherwise would be. Second, that when trend followers are done stopping out, there could be a snap back, as the prices had been artificially depressed by the stops.

Perhaps we could model this and see if there is any money in exploiting such a phenomenon.

Counter Model Logic

The general idea here is to reverse the logic of trend following models. What we want to do is to figure out more or less where trend models stop out, and then enter positions about that time. This mean reversion style model will try to enter during those pullbacks, betting on the price moving up again. Clearly, with this type of model, the entry timing is the

most important element.

When exploring trading ideas, it can often help to try to isolate the part of the strategy that appears most important, as an initial test. That's what we are going to try to do here. We are going to explore if there is a predictive value to enter into the kind of pullbacks where trend followers usually stop out.

As with all trading models in this book I want to again stress, once again, that I'm trying to teach and demonstrate concepts. These are not production grade models and they are not meant to be. The point here is for you to learn how to construct models, and giving you exact rules to copy and trade would very much defeat that purpose. Study these models, learn from them and built your own production grade models.

The parameters are not optimized or selected to show the best possible result. They are quite deliberately a bit of middle of the road kind of settings. They are picked more or less randomly, from a range of reasonable values.

So, having gotten that out of the way, now take a look at the entry rules. We are only looking to buy dips in bull markets. In my view, the dynamics of bear market rallies tend to be so different from bull market dips that we can't use the same logic, or at least not the same settings. In order to keep things reasonably simple, we are going to focus on bull market dips.

That means that we need to know if there is a bull market on. We need to define what a bull market is.

I will define a bull market here as when the 40 day exponential moving average is above the 80 day exponential moving average. Exponential moving averages are quite popular among technical analysts as they are said to react faster. What they do is to up-weight recent observations compared to older ones. I use it here, not because it's better, but because it can be a useful tool and sooner or later someone will come around to asking how to calculate exponential moving average in Python.

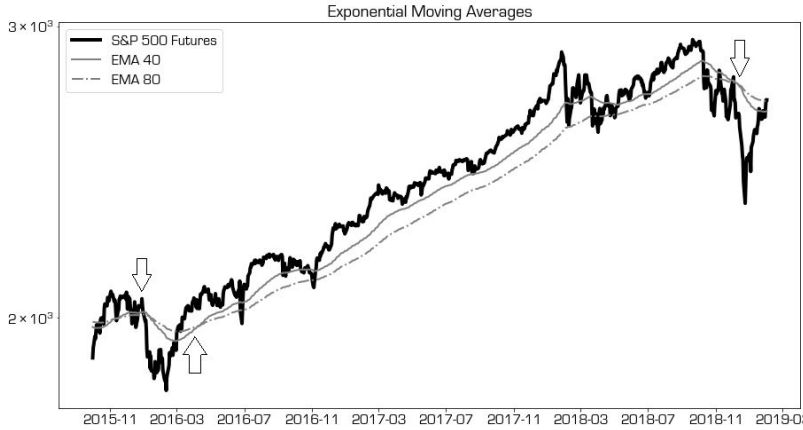


Figure 17-1 Exponential Moving Average

There is a very important thing to understand when working with exponential moving averages though. This is a classic source of confusion with many technical analysis indicators. The age old question about why your exponential moving average is not the same as mine. While it may sound as if a 40 day exponential moving average only uses and only needs 40 daily closes, that's not the case. The exponential weighting starts with the previous value, and that means that it never really drops data points. It merely down-weights them into oblivion over time.

What is important to understand here is that you will get a slightly

different value for a 40 day exponential moving average if you apply it on a half year time series history for a market, than if you apply it on ten years' worth of history for the same market. I guess that what I'm trying to say here is, please refrain from sending me emails about why your EMA is a little different from mine.

Quantifying Pullbacks

Once we know that we are in a bull market, we will look for pullbacks. An easy way to quantify pullbacks is to measure the distance from a recent high, but this needs to be put into some sort of a context. Clearly it wouldn't make sense to trigger a trade on a pullback of ten dollars for instance, as a ten dollar pullback in gold is quite different from the same in gasoline. Neither would percentage pullbacks make any sense, given the vastly varying volatility across the futures markets.

No, what we need to do is to standardize the pullback to the volatility of each market. If you think about it, we already have a good tool for this. We have been using standard deviation as a volatility proxy in previous models for the purpose of position sizing. There is nothing stopping us from reusing this analytic for standardizing pullbacks.

So in this model we will use a 40 day standard deviation of price changes both for position sizing purposes and for pullback. A pullback will be measured as the difference between current price and the highest close in the past twenty days, divided by the standard deviation. That will tell us how many standard deviations we are away from the highest price in about a month.

This results in an analytic that can be compared across markets. You could use the same process for stocks, bonds, commodities and other markets, as it takes the volatility into account. In this model, we are going to buy to open if we are in a bull market, as defined by the moving averages described above, and we

see a pullback of three times the standard deviation.

In the Python code, we calculate this pullback with just a few lines. The snippet below is from the source code for the model, which you will find in its entirety later in this chapter. As you see, this code first checks if we have a positive trend. If so, we calculate the difference between the latest close value and the highest for 20 days back, the high_windo w variable, and divide this by the standard deviation.

```
        if h['trend'].iloc[-1]:
            pullback = (
              h['close'].values[-1] -
np.max(h['close'].values[-high_window:])
              ) / std
```

As for the exit, we are going to use a really simple logic, just to demonstrate the entry idea. I want to show that what is important with this kind of model is the entry. That does not mean that everything else is unimportant, but a mean reversion style model like this is very much dependent on a solid entry logic. That's not the case for many other types of

strategies.

To see if there is any predictive value in the entry method described, we are going to use two simple exit criteria. First, if the trend as defined by the two moving averages turns bearish, we exit on the following day. Second, if that does not happen we hold the position for 20 trading days, approximately one month. Then we exit.

You are probably wondering why there is no stop loss point and no target exit price here. Those things may very well make sense, and I encourage you to try it out. The goal here is to teach you concepts and provide ideas for further research. Replicate what is shown here, try it out, modify it and make it your own.

Rules Summary

Long positions are allowed if the 40 day exponential moving average is above the 80 day exponential moving average. If the price in a bull market falls back three times its standard deviation from the

highest closing price in the past 20 days, we buy to open. If the trend turns bearish, we exit. If the position has been held for 20 trading days, we exit. Position size is volatility parity, based on standard deviation.

Counter Trend Source Code

In the previous model, the time return model, we learnt how to make a dynamically updating graph during the backtest, showing the portfolio value as it is being calculated. This time around, I will keep that, but also add an updating chart to show how the exposure changes over time.

```
import zipline
from zipline.api import future_symbol,  \
    set_commission, set_slippage, schedule_function,
date_rules, \
    time_rules, continuous_future, order_target
from datetime import datetime
import pytz
import pyfolio as pf
import pandas as pd
```

```
import numpy as np

from zipline.finance.commission import PerTrade,
PerContract
from zipline.finance.slippage import FixedSlippage,
VolatilityVolumeShare

# These lines are for the dynamic text reporting
from IPython.display import display
import ipywidgets as widgets
out = widgets.HTML()
display(out)

"""
Model Settings
"""
starting_portfolio = 20000000
vola_window = 40
slow_ma = 80
fast_ma = 40
risk_factor = 0.0015
high_window = 20
days_to_hold = 20
dip_buy = -3

def report_result(context, data):
    context.months + = 1
    today = zipline.api.get_datetime().date()
    # Calculate annualized return so far
    ann_ret =
np.power(context.portfolio.portfolio_value /
starting_portfolio,
                    12 / context.months) - 1
```

```
    # Update the text
    out.value = """{} We have traded <b>{}</b>
months
    and the annualized return is <b>{:.2%}</
b>""".format(today, context.months, ann_ret)

def initialize(context):
    """
    Cost Settings
    """
    context.enable_commission = True
    context.enable_slippage = True
```

```
    if context.enable_commission:
        comm_model = PerContract(cost=0.85,
exchange_fee=1.5)
    else:
        comm_model = PerTrade(cost=0.0)
    set_commission(us_futures=comm_model)
```

```
    if context.enable_slippage:

slippage_model=VolatilityVolumeShare(volume_limit=0.3)
    else:
        slippage_model=FixedSlippage(spread=0.0)
```

```
    set_slippage(us_futures=slippage_model)
```

```
    agricultural = [
        'BL',
        'CC',
        'CT',
```

```
    'FC',
    'KC',
    'LB',
    'LR',
    'OJ',
    'RR',
    '_S',
    'SB',
    'LC',
    'LS',
]
nonagricultural = [
    'CL',
    'GC',
    'HG',
    'HO',
    'LG',
    'PA',
    'PL',
    'RB',
    'SI',
    'NG',
    'LO',
]
currencies = [
    'AD',
    'BP',
    'CD',
    'CU',
    'DX',
    'NE',
    'SF',
    'JY',
]
```

```
    equities = [
        'ES',
        'NK',
        'NQ',
        'YM',
    ]
    rates = [
        'ED',
        'FV',
        'TU',
        'TY',
        'US',
    ]
```

```
    markets = agricultural + nonagricultural +
currencies + equities + rates
```

```
    context.universe = \
        [
        continuous_future(market, offset=0,
roll='volume', adjustment='mul') \
        for market in markets
        ]
```

```
    # Dictionary used for keeping track of how many
days a position has been open.
    context.bars_held = {market.root_symbol: 0 for
market in context.universe}
```

```
    # Schedule daily trading
    schedule_function(daily_trade,
date_rules.every_day(), time_rules.market_close())
```

```
    # We'll just use this for the progress output
    # during the backtest. Doesn't impact anything.
    context.months = 0

    # Schedule monthly report output
    schedule_function(
        func=report_result,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )

def roll_futures(context, data):
    open_orders = zipline.api.get_open_orders()

    for held_contract in context.portfolio.positions:
        # don't roll positions that are set to change by
core logic
        if held_contract in open_orders:
            continue

        # Save some time by only checking rolls for
        # contracts expiring in the next week
        days_to_auto_close = (
            held_contract.auto_close_date.date() -
data.current_session.date()
        ).days
        if days_to_auto_close > 5:
            continue

        # Make a continuation
        continuation = continuous_future(
```

```
            held_contract.root_symbol,
            offset=0,
            roll='volume',
            adjustment='mul'
            )
```

```
        # Get the current contract of the continuation
        continuation_contract =
data.current(continuation, 'contract')
```

```
        if continuation_contract != held_contract:
            # Check how many contracts we hold
            pos_size =
context.portfolio.positions[held_contract].amount
            # Close current position
            order_target(held_contract, 0)
            # Open new position
            order_target(continuation_contract, pos_size)
```

```
def position_size(portfolio_value, std, pv):
    target_variation = portfolio_value * risk_factor
    contract_variation = std * pv
    contracts = target_variation / contract_variation
    # Return rounded down number.
    return int(np.nan_to_num(contracts))

def daily_trade(context, data):
```

```
    open_pos = {pos.root_symbol: pos for pos in
context.portfolio.positions}
```

```
    hist = data.history(
```

```
        context.universe,
        fields=['close', 'volume'],
        frequency='1d',
        bar_count=250,
    )
```

```
    # Calculate the trend
    hist['trend'] =
hist['close'].ewm(span=fast_ma).mean() >
hist['close'].ewm(span=slow_ma).mean()
```

```
    for continuation in context.universe:
        root = continuation.root_symbol
```

```
        # Slice off history for this market
        h = hist.xs(continuation, 2)
```

```
        # Calculate volatility
        std = h.close.diff()[-vola_window:].std()
```

```
        if root in open_pos: # Check open positions first.
            context.bars_held[root] += 1 # One more day
held
```

```
            if context.bars_held[root] >= 20:
                # Held for a month, exit
                contract = open_pos[root]
                order_target(contract, 0)

            elif h['trend'].iloc[-1] == False:
                # Trend changed, exit.
```

```
            contract = open_pos[root]
            order_target(contract, 0)

        else: # Check for new entries
            if h['trend'].iloc[-1]:
```

```
                # Calculate the pullback
                pullback = (
                    h['close'].values[-1] -
np.max(h['close'].values[-high_window:])
                ) / std
```

```
                if pullback < dip_buy:
                    # Get the current contract
                    contract = data.current(continuation,
'contract')
```

```
                    # Calculate size
                    contracts_to_trade = position_size( \

context.portfolio.portfolio_value, \
                                        std, \

contract.price_multiplier)
                    # Trade
                    order_target(contract, contracts_to_trade)
```

```
                    # Reset bar count to zero
                    context.bars_held[root] = 0
```

```
    # Check if we need to roll.
    if len(open_pos) > 0:
```

```
        roll_futures(context, data)

start = datetime(2001, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2018, 12, 31, 8, 15, 12, 0, pytz.UTC)

perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    capital_base=starting_portfolio,
    data_frequency = 'daily',
    bundle='futures' )
```

Counter Trend Results

Once again we see that a simple set of rules can show pretty good results, as long as there is a solid underlying premise. There are a few worrying negative months, as seen in Table 17.1, but given the wonky stop logic in this demo model that shouldn't be too surprising. What we see is a strategy that can get pretty volatile at times, but which has delivered strong risk adjusted returns over time.

Not bad for such a simple model.

The takeaway here is that there seem to be a value in this kind of entry logic. The conclusion is not that you should run to your broker and bet all of your cash on this, but rather that this could be an interesting research topic to develop further. If this simple logic can result in pretty interesting returns, surely you can make it better, adapt it to your own needs, add some features to it and create a proper production grade model.

Table 17.1 - Counter Trend Monthly Returns

Ye	Year
2001	+-3.8
20	+41.2
2003	+-19.8
20	+-16.1
2005	+-17.8
20	-6.4
2007	+-30.8
20	+19.3
2009	+-42.8
20	+19.2
2011	+-13.8
20	+7.9
2013	+-9.8

20	+24.0
2015	+[illegible]
20	+11.1
2017	+[illegible]

The equity curve in Figure 17-2 shows us a strategy with a clear positive bias. It tends to have a fairly low correlation to the equity markets, and it usually underperforms during long term bull markets. That's not necessarily a problem though, as it shows strong performance during bear markets, and a substantial outperformance in the long run.

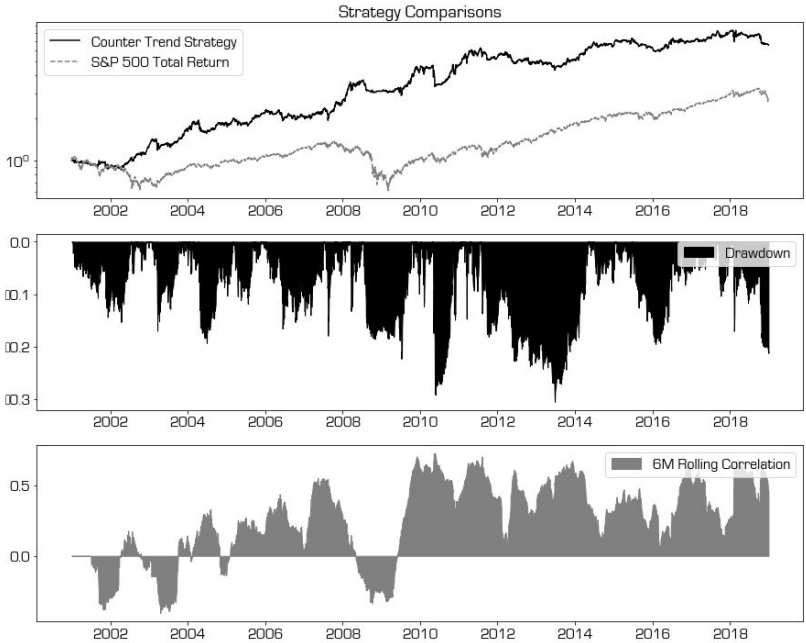


Figure 17-2 Counter Trend Simulation Results

In Table 17.2 you can see what would have happened if you started this strategy at a given year, and kept trading it for a number of years from that point. This can be a useful way to quickly get an overview of how the initial staring time sometimes can matter quite a lot for a trading model. The numbers show annualized return over the number of years displayed on the x-axis.

While the long term return looks attractive, this kind of graph can quickly

show us that there are starting years where you may not have been as happy. If you would have started this strategy in 2011, you would only have annualized about 4% after 8 years. But on the other hand, had you started in 2007, you would have seen an annualized gain of 16% after the same number of years.

Table 17.2 Holding period analysis for the counter trend model

+	2001
+	
+	2003
+	
+	2005
+	
+	2007
+	
+	2009
+	
+	2011
+	2013
+	
	2015
+	
+	2017

This model clearly showed lower returns in the past few years. The final year of the backtest certainly had an impact, as it ended at about 15 percent in the red. Some readers may be surprised about this, as books tend to show models that performed remarkably well for the entire period, or at the very least the most recent years. It would have been easy for me to tweak the settings of the backtest until such a result was achieved, but that would really be contrary to the brief of this book.

If you want to curve fit this backtest, go ahead and tweak the settings until you get the results you like to see. Just keep in mind that it's not terribly likely to result in any meaningful predictive value. The settings used here are picked not based on the backtest results they yield, but for their symmetry to the trend model shown earlier. The reason for that, is simply that it makes it easier to explain the underlying rationale of this trading approach.

While curve fitting is a bad idea, that does not mean that you should go ahead and trade these exact settings either. What you need to do is think long and hard about what you want to achieve with a model. It's not just a matter of aiming for high returns. It's a matter of finding a desired return profile, and that does not always mean highest possible returns. In fact, it very rarely means that.

No, a much more interesting way to see a mean reversion model like this, is as a portfolio component, traded alongside with trend following models. But I'm getting ahead of myself. More on that later on.

18 Trading the Curve

When I set out to write this book, my primary stated purpose was to broaden the horizon of retail traders. To show things that they might not be aware

of, to make them see new possibilities and ways of working that they might not otherwise have thought of. Hopefully the concept in this chapter will be such a lesson, showing a type of trading which many readers probably had not considered.

What I intend to demonstrate here is a method of trading futures without using any historical data. Yes, we will still use systematic, algorithmic trading, but we won't need any price history for the trading logic. All we need is the current prices.

We are going to calculate the cost of carry and use this as the sole input for trade selection. This means that we need to look at not just the front contract, but at contracts further out. Most futures based trading models trade only the front contract, and nothing else. For this model here, we won't trade the front at all, only further out points.

The term I use here is curve trading, and that's how I refer to this trading approach. To be clear, what we will be

doing here is to trade carry. The cost of carry is implied by the shape of the term structure curve. Depending on what terminology you are used to, you may be of the view that curve trading means taking offsetting positions on the curve, long one month and short another. That's a highly related concept, and if you understand the model presented in this chapter you can easily expand to such calendar spread type models, as they are often called, later on. But for now, we are simply trading carry.

Term Structure Basics

Some readers already know exactly where I'm heading with this, but others may prefer to get a little refresher course on term structures first.

Futures contracts have a limited life span. There is a date when each contract ceases to exist. This means that there is not just one futures gold contract, but a rather large amount of them. At any given time, there will be one specific contract

that has the highest liquidity. Most people would trade the contract that's closest to expiry, as long as it has not passed the first notice date.

The first notice date, usually a month before expiry, refers to the date when a contract becomes deliverable, and someone still holding it can be called upon to make or take delivery. As traders, we don't really want to be part of that, but it's not really a risk in reality. Your broker won't let you keep positions open after that date anyhow. First notice date is a concept primarily in commodity futures, and that's the only sector which we are going to trade in this chapter. It's in the commodity sector that term structure trading is of most interest.

Figure 18-1 shows how the term structure for soybeans looked at the time of writing this. As with any financial data, this is ever changing of course. This figure shows the expiry dates on the x-axis, and corresponding price and open interest for each contract. These are the contracts currently trading, again at the time of writing this.

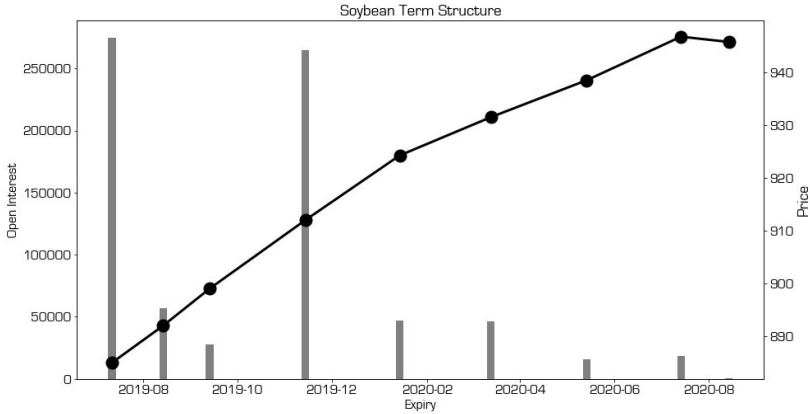


Figure 18-1 Term Structure in Contango

In this example, you can see that each consecutive point on the curve is a little higher up than the previous one. That's, contracts are more expensive the longer time there is until they expire. That situation is called **contango** . The reversed, if each point would get cheaper, would be called **backwardation** , which is what you see in Figure 18-2.

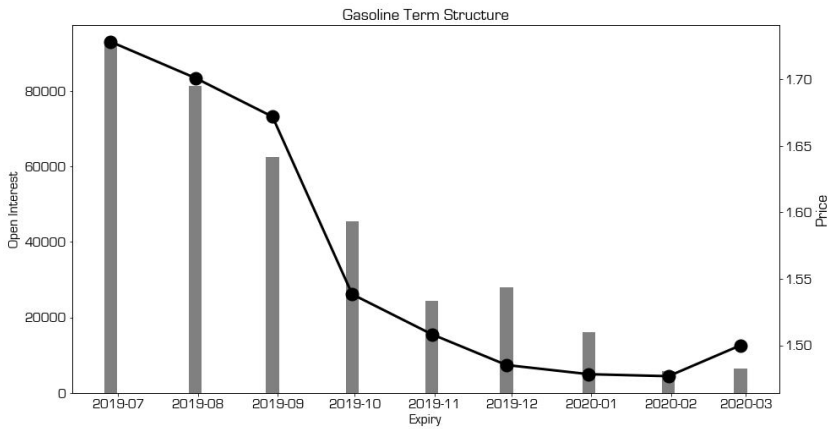


Figure 18-2 Gasline in Contango

The closest contracts tends to be traded fairly close to the underlying asset. The less time left to expiry, the closer the price tends to be. The reason for that is simple. At expiry time, the value of the contract is the same as the value of the underlying, as it will be settled for it. But when there is plenty of time left, other factors take the driver's seat.

There are multiple reasons for why the price further out in the future is different, but as quantitative traders we rarely need to dig deep into these reasons. Factors in play include interest rates, cost of storage and seasonality.

What matters is how to interpret this pattern. Here is a simple way to think about it. If the term structure is in contango, as in Figure 18-1, there is a bearish bias. Why? Because the closer a contract gets to expiry, the closer the price will be to the underlying. So if the underlying stays exactly the same, each point on the curve in Figure 18-1 would have to slowly move down until it finally

hits the underlying asset's price.

The reverse is therefore true for backwardation, which then has a built in bullish bias. The underlying would need to move down simply for the points on the backwardation structure to stay the same. If the underlying does not move, the contracts need to come up to meet it.

Now if you understand the general idea, you are probably already thinking about how to quantify this effect.

Quantifying Term Structure Effect

While there are different ways to quantify the term structure, I will demonstrate a methodology which I believe makes intuitive sense. This involves calculating an implied annual yield, or cost of carry if you will. It needs to be annualized, so that the resulting number can be compared across different delivery dates. As always in finance, time matters. If a contract three months out is

traded at a 2% discount that is more significant than if a contract 12 months out is traded at the same 2% discount. Just like how it's preferable to make a hundred dollars today than in a year from now.

We will use the same data as for Figure 18-1 in this example. First I will show you how to calculate one point manually and then we will look at how to get this done easily for the entire curve.

The first point in the curve, the closest contract, expires on the 14 th of March 2019. That contract, the SH9, is currently traded at 907.50 cents per bushel. No, you don't really need to know much about bushels to trade soybeans. The next contract out is the SK9, expiring on May 14 th the same year, and that's traded at 921.50 cents per bushel.

The SK9 expires 61 days after SH9. If the underlying, cash soybean, does not move at all, the SK9 would need to move from 921.50 down to 907.50 in the next 61 days. That makes for a loss of 1.52%. A loss of 1.52% in 61 days, would be the

equivalent of an annualized loss of 8.75%.

$$(((-0.0152+1)^{(365/61)})-1)=-8.75\%$$

This gives us a number that we can both relate to and compare across markets and expiration dates. We have a quantifiable, comparable yearly yield number.

Getting this sort of thing done quickly on a futures chain is very simple. First we will start off with a standard Pandas **DataFrame** with the prices and expiry dates. The data in Table 18.1 is the same as we used for the term structure figure just earlier, and it shows a market in contango. If you did not skip the Python introduction in the first part of this book, you should have no problem getting this data into a **Pandas DataFrame** .

Table 18.1 Term Structure Data

price	expiry	interest
104	March 2019	937,419
114	May 2019	921,954
122	July 2019	935,934
134	August 2019	940,972
143	September 2019	948,429

54	952.65	6	13	2019					
64	961.50	7	20	2020					

Assuming you have the above table in a **DataFrame** called simply `df`, you could get the analytic we are looking for step by step like this.

```
df['day_diff'] = (df['expiry'] - df.iloc[0]['expiry']) /
np.timedelta64(1, 'D')
df['pct_diff'] = (df.iloc[0].price / df.price) - 1
df['annualized_carry'] = (np.power(df['pct_diff'] + 1,
(365 / df.day_diff))) – 1
```

As you should have figured out about Python at this point, you could of course do all of this in a single row if you like. The ability of Python to perform multiple complex operations in a single row can at times be great, but it also tends to make the code more difficult to follow.

Running this code, and adding these columns should result in a **DataFrame** like the one below in Table 18.2. Now we have something useful. This is a table that we can use directly for trading logic.

Table 18.2 Annualized Carry

| | | |

We are going to make quite a simple model, based on the logic just described. We are going to trade only commodities, and only a set of fairly liquid ones. This is not a strategy that can be used in any market, and it would be little use trying the same methodology on currency futures or low liquidity markets.

In this version, we attempt to find a contract one year out on the curve to trade. That is, a contract expiring one year later than the current front contract. Trading is only done once per week at which point we go over each market we cover, identify the contract one year out and calculate the annualized carry.

Having calculated this for all the markets we cover, the model sorts the markets on the carry and picks the top 5 backwardation to go long and the top 5 contango to go short.

By default here, we attempt to trade an equal notional amount of each contract. Why, you may ask, after my previous rants on this book on the value of volatility parity allocation. Two

reasons. First, I figured that I may have already confused readers enough with this very different way of constructing a model and I want to keep the logic simple and easy to follow. And second, for this particular approach, it's not going to make a whole lot of difference. We are only trading one asset class and the volatility differences won't be enormous.

As opposed to other models in this book, we are now dealing with potentially low liquidity markets, and that requires some extra consideration. Even if the gold front contract is heavily traded, the contract one year out will probably not be.

This means that we need to ensure that our model, or rather our backtester, can handle it realistically. We simply can't assume that we will get filled on open on any size. Luckily, Zipline is pretty good at this, and the model will use a slippage condition to ensure that we never trade more than 20% of daily volume.

The model will then take a maximum of 20% of the volume, and

leave the remainder of the open order for the following trading day. Large orders will be spread out over several days, when needed.

But I will also put in one extra liquidity condition, to be on the safe side. Consider if for instance the coffee contract one year out has an average daily volume of 100 contracts, and your model decides it's a good idea to place an order for 1,000 contracts. The Zipline logic will model the fills each day, and it may take you a couple of weeks or more to get completely filled. Realistic perhaps, but that's not what we want to see here.

For that reason, we'll put a condition to check the average daily volume on the contract we want to trade, and refrain from placing any order larger than 25% of that. It may still take a couple of days sometimes to get filled, but it shouldn't take more. This condition will limit the maximum size of the orders.

As you will see in the source code, the default version that I'm showing you here will attempt to build an exposure of

150% long and 150% short, for a near zero net exposure. Of course, you can't expect to be "delta neutral" by for instance having a short soybean position offsetting a long crude oil position. The term delta neutral stems from the options markets and means that you, at least in theory, lack a directional exposure to the markets.

Don't expect a perfect hedge of any kind, but of course it does help a bit with the overall risk to have a long and a short leg. All in all, the gross exposure level of 300% is not out of proportion for a reasonably diversified futures model.

As previously, all major settings are up top in the code, so that you can easily try your own variations once you have set up your local Python modelling environment.

Curve Trading Source Code

By now you should be quite

familiar with the general structure of Zipline backtest code. I will show you the key parts of this particular model first, and then as usual, at the end of this section you will find the complete source code.

First up, the settings for the model.

```
# settings
spread_months = 12
pos_per_side = 5
target_exposure_per_side = 1.5
initial_portfolio_millions = 1
volume_order_cap = 0.25
```

With these settings we can switch things up a bit and easily try variations. The first setting, spread_month s , sets how far out on the curve we are targeting. In the default case, we're looking for contracts a full month out.

The second setting, pos_per_sid e , is for now many markets do we want to be long, and how many we want to be short. In the current implementation, this number sets both long and short positions.

Next setting, target_exposure_per_sid e , sets percent exposure per side. At the default value of 1.5, we are trying to build

an exposure of 150% long and 150% short.

Next we set how many millions we want to start off with, and lastly we set a volume order cap. That last setting can be quite important for this particular model. Trading these less liquid contracts with longer to expiry means that we need to be careful with liquidity. We are not able to trade too large, and we should always make sure that there is enough capacity.

At the default 0.25 value for volume_order_ca p , our trading code will limit orders to 25% of the average daily volume. This will make sure that we don't try to place orders far larger than the market can handle, and end up spending weeks trying to get it executed.

The trading logic is done once a week, which is the only time when we are adjusting our portfolio composition. Since we are trading specific contracts here, and not trying to stay with the front contract as is otherwise common, there is no need for roll logic.

Let's look at the logic step by step,

to see what operations are performed each month for this model.

First we merely create an empty **DataFrame** , using the list of markets in our investment universe. We're going to populate this **DataFrame** with our calculated carry analytics in a moment.

```
def weekly_trade(context, data):
    # Empty DataFrame to be filled in later.
    carry_df = pd.DataFrame(index = context.universe)
```

Once we have that empty **DataFrame** , we are going to loop through all the markets we cover, get the front contract and the contract one year out, calculate the carry and store it in the **DataFrame** . Here is it, step by step.

This first part starts the loop, gets the contract chain and transforms it into a **DataFrame** with contracts and corresponding expiry dates.

```
    for continuation in context.universe:
        # Get the chain
        chain = data.current_chain(continuation)

        # Transform the chain into dataframe
        df = pd.DataFrame(index = chain)
        for contract in chain:
            df.loc[contract, 'future'] = contract
```

```
        df.loc[contract, 'expiration_date'] =
contract.expiration_date
```

Next we locate the contract closest to our target date. As the default setting here is to trade 12 months out, we're looking for a contract expiring in one year. While it is quite probable that we'll find a contract at 12 months out, it is not certain that all markets will have a contract 3 or 9 months out, and therefore this logic to ensure that the model can handle such variations.

```
    closest_expiration_date =
df.iloc[0].expiration_date
    target_expiration_date = closest_expiration_date
+ relativedelta(months=+spread_months)
    df['days_to_target'] = abs(df.expiration_date -
target_expiration_date)
    target_contract = df.loc[df.days_to_target ==
df.days_to_target.min()]
```

The contract that we located there, `target_contrac t` , is the one expiring closest to what we're looking for. The logic above first checks what exact date it is X months from the front contract expiry, where X is 12 by default. Then we make a column for all contracts in the chain showing how many days difference they have to that

target date. Finally, we pick the contract with the least difference.

Now we need to get the last closing prices for the front contract as well as the target contract, and calculate the annualized carry.

```
    # Get prices for front contract and target contract
    prices = data.current(
        [
            df.index[0],
            target_contract.index[0]
        ],
        'close'
    )
```

```
    # Check the exact day difference between the
contracts
    days_to_front = int(
        (target_contract.expiration_date -
closest_expiration_date)[0].days
    )
```

```
    # Calculate the annualized carry
    annualized_carry = (np.power(
                        (prices[0] / prices[1]), (365 /
days_to_front))
                        ) - 1
```

If you've paid attention so far in the book, you may have noticed that the formula for calculating annualized yield is

a little familiar. It is of course the same logic as we used to calculate annualized momentum scores in chapter 12.

Now we have the data that we need to populate that **DataFrame** we created at the start. In order to compare all the markets, and pick the most attractive carry situations, we need to know which one is the front contract, which is the target contract, and what the annualized carry is.

```
        carry_df.loc[continuation, 'front'] =
df.iloc[0].future
        carry_df.loc[continuation, 'next'] =
target_contract.index[0]
        carry_df.loc[continuation, 'carry'] =
annualized_carry
```

Now sort them, and get rid of any potentially empty rows.

```
    # Sort on carry
    carry_df.sort_values('carry', inplace=True,
ascending=False)
    carry_df.dropna(inplace=True)
```

This **DataFrame** object, called `carry_df` in the code, now holds the analytics we need to decide what to trade. Now we can start by picking the top and

bottom five contracts, the ones we should be long and short.

```
# Contract Selection
for i in np.arange(0, pos_per_side):
    j = -(i+1)

    # Buy top, short bottom
    long_contract = carry_df.iloc[i].next
    short_contract = carry_df.iloc[j].next

    new_longs.append(long_contract)
    new_shorts.append(short_contract)
```

What this code does is to loop through the numbers 0 through 4, picking five contracts from the top and five from the bottom of the deck. Now that we know which contracts we want to be long and which we want to be short, all we need to do is to figure out how much of each to trade and then execute.

In order to figure this out, we need to pull some historical data. As we said earlier, we are going to limit our orders to 25% of average daily volume, so obviously we need to know what the average daily volume is.

```
# Get data for the new portfolio
```

```
    new_portfolio = new_longs + new_shorts
    hist = data.history(new_portfolio,
fields=['close','volume'],
        frequency='1d',
        bar_count=10,
        )
```

We also said we are doing an equal weighted allocation. If you are inclined to tinkering, the allocation part is something you may want to work further with on your own.

```
    # Simple Equal Weighted
    target_weight = (
        target_exposure_per_side  *
context.portfolio.portfolio_value
        ) / pos_per_side
```

At this point, we are all set to loop through the markets which we want to hold for the next week. Here's what we're going to do. For each market, we calculate target contracts to hold, enforce a limit based on the average volume and execute.

```
    # Trading
    for contract in new_portfolio:
        # Slice history for contract
        h = hist.xs(contract, 2)


        # Equal weighted, with volume based cap.
        contracts_to_trade = target_weight / \
```

```
            contract.price_multiplier / \
            h.close[-1]
```

```
        # Position size cap
        contracts_cap = int(h['volume'].mean() *
volume_order_cap)
```

```
        # Limit trade size to position size cap.
        contracts_to_trade = min(contracts_to_trade,
contracts_cap)
```

```
        # Negative position for shorts
        if contract in new_shorts:
            contracts_to_trade *= -1
```

```
        # Execute
        order_target(contract, contracts_to_trade)
```

That leaves only one final part of housekeeping. Can you think of what we didn't do yet?

We analyzed the markets, we selected contracts, we constructed a new portfolio, calculated position sizes and we traded. But we didn't close the old positions yet. The only thing that's left is to loop through all open positions, and close those that are not part of next week's portfolio.

```
    # Close any other open position
    for pos in context.portfolio.positions:
        if pos not in new_portfolio:
            order_target(pos, 0.0)
```

As with the previous models, I will show the complete model code used here below. As you will see, I used the dynamically updating chart during the simulation run, as we have seen earlier. This time, I added a drawdown chart, just to show how that is done.

```
%matplotlib notebook

import zipline
from zipline.api import future_symbol,  \
    set_commission, set_slippage, schedule_function, \
    date_rules, time_rules, continuous_future,
order_target

from datetime import datetime
import pytz
import matplotlib.pyplot as plt
import pyfolio as pf
import pandas as pd
import numpy as np

from zipline.finance.commission import PerTrade,
PerContract
from zipline.finance.slippage import FixedSlippage,
VolatilityVolumeShare

# We'll use this to find a future date, X months out.
```

```
from dateutil.relativedelta import relativedelta

# settings
spread_months = 12
pos_per_side = 5
target_exposure_per_side = 1.5
initial_portfolio_millions = 1
volume_order_cap = 0.25



# DataFame for storing and updating the data that we
want to graph
dynamic_results = pd.DataFrame()

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(211)
ax.set_title('Curve Trading')
ax2 = fig.add_subplot(212)
ax2.set_title('Drawdown')

def initialize(context):
    """
    Friction Settings
    """
    context.enable_commission = True
    context.enable_slippage = True


    if context.enable_commission:
        comm_model = PerContract(cost=0.85,
exchange_fee=1.5)
    else:
        comm_model = PerTrade(cost=0.0)
    set_commission(us_futures=comm_model)
```

```
    if context.enable_slippage:

slippage_model = VolatilityVolumeShare(volume_limit = 0.3)
    else:
        slippage_model = FixedSlippage(spread = 0.0)
    set_slippage(us_futures = slippage_model)
```

```
    """
    Markets to trade
    """
    most_liquid_commods  =  [
        'CL','HO','RB','NG','GC','LC','_C','_S','_W','SB', 'HG',
'CT', 'KC'
    ]
```

```
    context.universe  =  [
        continuous_future(market, offset = 0,
roll = 'volume', adjustment = 'mul')
        for market in most_liquid_commods
        ]
```

```
    schedule_function(weekly_trade,
date_rules.week_start(), time_rules.market_close())
```

```
schedule_function(update_chart,date_rules.month_start(),
time_rules.market_close())
```

```
def update_chart(context,data):
    # This function continuously update the graph
during the backtest
    today  =  data.current_session.date()
```

```
    pv = context.portfolio.portfolio_value
    exp = context.portfolio.positions_exposure
    dynamic_results.loc[today, 'PortfolioValue'] = pv
```

```
    drawdown = (pv /
dynamic_results['PortfolioValue'].max()) - 1
    exposure = exp / pv
    dynamic_results.loc[today, 'Drawdown'] =
drawdown
```

```
    if ax.lines:
        ax.lines[0].set_xdata(dynamic_results.index)

ax.lines[0].set_ydata(dynamic_results.PortfolioValue)
        ax2.lines[0].set_xdata(dynamic_results.index)

ax2.lines[0].set_ydata(dynamic_results.Drawdown)
    else:
        ax.plot(dynamic_results.PortfolioValue)
        ax2.plot(dynamic_results.Drawdown)
```

```
    ax.set_ylim(
        dynamic_results.PortfolioValue.min(),
        dynamic_results.PortfolioValue.max()
    )
    ax.set_xlim(
        dynamic_results.index.min(),
        dynamic_results.index.max()
    )
    ax2.set_ylim(
        dynamic_results.Drawdown.min(),
        dynamic_results.Drawdown.max()
    )
```

```
    ax2.set_xlim(
        dynamic_results.index.min(),
        dynamic_results.index.max()
    )
```

```
    fig.canvas.draw()


def weekly_trade(context, data):
    # Empty DataFrame to be filled in later.
    carry_df = pd.DataFrame(index = context.universe)
```

```
    for continuation in context.universe:
        # Get the chain
        chain = data.current_chain(continuation)

        # Transform the chain into dataframe
        df = pd.DataFrame(index = chain)
        for contract in chain:
            df.loc[contract, 'future'] = contract
            df.loc[contract, 'expiration_date'] =
contract.expiration_date

        # Locate the contract closest to the target date.
        # X months out from the front contract.
        closest_expiration_date =
df.iloc[0].expiration_date
        target_expiration_date = closest_expiration_date
+ relativedelta(months=+spread_months)
        df['days_to_target'] = abs(df.expiration_date -
target_expiration_date)
        target_contract = df.loc[df.days_to_target ==
df.days_to_target.min()]
```

```
        # Get prices for front contract and target contract
        prices = data.current(
            [
                df.index[0],
                target_contract.index[0]
            ],
            'close'
        )
```

```
        # Check the exact day difference between the
contracts
        days_to_front = int(
            (target_contract.expiration_date -
closest_expiration_date)[0].days
        )
```

```
        # Calculate the annualized carry
        annualized_carry = (np.power(
                            (prices[0] / prices[1]), (365 /
days_to_front))
                            ) - 1
```

```
        carry_df.loc[continuation, 'front'] =
df.iloc[0].future
        carry_df.loc[continuation, 'next'] =
target_contract.index[0]
        carry_df.loc[continuation, 'carry'] =
annualized_carry
```

```
    # Sort on carry
    carry_df.sort_values('carry', inplace=True,
```

```
ascending=False)
    carry_df.dropna(inplace=True)
```

```
    new_portfolio = []
    new_longs = []
    new_shorts = []
```

```
    # Contract Selection
    for i in np.arange(0, pos_per_side):
        j = -(i+1)
```

```
        # Buy top, short bottom
        long_contract = carry_df.iloc[i].next
        short_contract = carry_df.iloc[j].next
```

```
        new_longs.append(long_contract)
        new_shorts.append(short_contract)

    # Get data for the new portfolio
    new_portfolio = new_longs + new_shorts
    hist = data.history(new_portfolio,
fields=['close','volume'],
        frequency='1d',
        bar_count=10,
        )
```

```
    # Simple Equal Weighted
    target_weight = (
        target_exposure_per_side *
context.portfolio.portfolio_value
        ) / pos_per_side
```

```
    # Trading
    for contract in new_portfolio:
        # Slice history for contract
        h = hist.xs(contract, 2)

        # Equal weighted, with volume based cap.
        contracts_to_trade = target_weight / \
            contract.price_multiplier / \
            h.close[-1]

        # Position size cap
        contracts_cap = int(h['volume'].mean() *
volume_order_cap)

        # Limit trade size to position size cap.
        contracts_to_trade = min(contracts_to_trade,
contracts_cap)

        # Negative position for shorts
        if contract in new_shorts:
            contracts_to_trade *= -1

        # Execute
        order_target(contract, contracts_to_trade)

    # Close any other open position
    for pos in context.portfolio.positions:
        if pos not in new_portfolio:
            order_target(pos, 0.0)

start = datetime(2001, 1, 1, 8, 15, 12, 0, pytz.UTC)
end = datetime(2018, 12, 30, 8, 15, 12, 0, pytz.UTC)
```

```
perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    capital_base=initial_portfolio_millions * 1000000,
    data_frequency = 'daily',
    bundle='futures' )
```

Curve Trading Results

This is an algorithmic trading model which does not use historical price series. That's rather unusual, and if you are not previously familiar with the concept of term structure or calendar trading, this may seem to be a very weird idea. But as you will see from the backtest results, even a simple implementation like this can actually work fairly well.

Glance through the monthly table first to get a feel for how the strategy behaves over time. For a model with no historical data, no stop losses, no targets, no indicators and simple equal weighting, it's really not all that bad.

Table 18.3 Curve Trading Monthly Returns

Year	[illegible]
20	+10.7
2002	+[illegible]
20	+8.1
2004	+[illegible]
20	+51.1
2006	+[illegible]
20	+24.2
2008	+[illegible]
20	+7.3
2010	+[illegible]
20	+23.9
2012	+[illegible]
20	+13.6
2014	-[illegible]
20	+19.4
2016	-[illegible]
20	+19.6
2018	+[illegible]

Looking at the equity curve in Figure 18-3 you see that it's surprisingly smooth. It shouldn't be surprising however that it shows no discernible correlation to the equity index. This strategy is very much a so called

uncorrelated strategy. That term is thrown around a lot in the industry, and refers to strategies which are at least in theory uncorrelated to the equity markets.

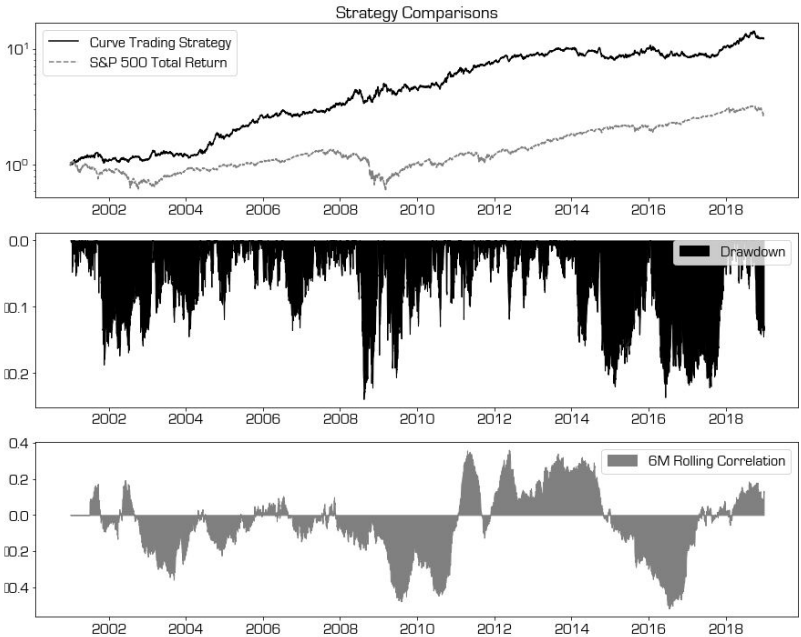


Figure 18-3 Curve Trading Returns

Table 18.4 shows the holding period returns, or what your percentage return would have looked like if you started this strategy by the beginning of a given year, and held for a certain number of years. What you see here are fairly good long term numbers for most starting points. Even picking the worst possible starting years, you see how recovery does

not take too long, and soon surpasses traditional investment approaches.

Table 18.4 Holding Period Returns - Curve Trading

Model Considerations

It can't be overly stressed that this

model is highly sensitive to liquidity. As you have seen in the source code for this model, only a single million was used as basis for trading. That's an exceedingly small amount to trade futures models with, but this is a type of model where you have a small guy advantage. It's possible to grind out quite high returns with lower capital deployments, but the game changes dramatically if you want to trade nine figures.

I do realize of course than many readers balk at my assortment that a million is a tiny bit of money. Of course it's an enormous amount of money, out in the real world. But as far as professional futures trading goes, it's really not much.

This model can be scaled up and down a bit. It can be traded with a bit less, and with a bit more, and still show interesting returns. But serious institutional size money would struggle.

This model implementation here is highly simple, again to teach a concept and demonstrate a way of approaching algorithmic modelling. The source code

made available to you here should enable you to experiment with your own versions of it, adapting and improving based on your own situation and requirements.

Generally speaking, you will find a bit better liquidity closer on the curve, to the left side. This implementation trades 12 months out, and there is usually quite limited trading there. If you trade for instance three months out, you will find a little better liquidity. Your slippage assumptions and preferences on how aggressive you want to trade will greatly impact your results.

Be careful with your modelling here. It would be very easy to change a couple of numbers in the code, and end up with a 50 percent per year or more return. But a simulation is only as good as the assumptions that go into it, and you are not likely to realize such returns in live trading.

Another area of research would be to look at combining this term structure information with other analytics. The approach shown here is purely looking at

the implied yield, or cost of carry, but there is no reason why your models would need to be so pure in real life. Yes, I'm just going to throw that out there and let you do the homework.

19 Comparing and Combining Models

On the futures side, we have now looked at a few different approaches. First, a standard trend model with trend filter, trailing stop, and breakout logic. Second, a simple time return model that only compares a monthly price with that of a year and half a year prior. Third, counter trend, or mean reversion approach which aims to enter when trend followers stop out, operating on a shorter time period. And finally a carry based trading model, looking only at the shape of the term structures curve.

We also started off with a

systematic equity momentum model, which trades only the long side of equities and should have quite a different return profile from absolute return futures models.

In the chapter about each of these models, you have probably noticed that I did not show the usual return statistics. That was very much on purpose, as I have come to realize that many readers stare too much as such figures, and miss the bigger picture. It's a bit like handing out slide printouts before a live presentation. No one is going to listen to what you have to say after that.

But now that you have presumably already worked your way through the previous chapters, it should be safe enough to show you the stats. The data you are looking for is in Table 19.1, where the strategies that we looked at earlier are listed, as well as the same statistics for the S&P 500 Total Return Index, all covering the backtesting period from the start of 2001 to the end of 2018.

Table 19.1 Futures Strategies Statistics

	[illegible]
trend	0.98
counter_trend	[illegible]
curve	1.22
time_return	[illegible]
syste[illegible]	0.76
SPXTR	[illegible]

Clearly the curve trading model is the best one, right? And the momentum isn't worth bothering with? Well, conclusions like that are the reason why I did not show these simple statistics earlier. Evaluating trading models is a more complex undertaking than simply looking at a table like this. You need to study the details, and study the long term return profile. And of course scalability. At the sharp end of the business, you often look for a specific behavior in the return profile, often relative to other factors. The answer to which model is more promising depends on what you happen to be looking for at the moment, and what would fit or complement your current portfolio of models.

All of these models are simple demo models. They are teaching tools, not production grade models. But they all show potential, and they can be polished up to become production grade models.

You can also see that all of them are orders of magnitudes more attractive than a buy and hold, stock market approach. Some readers may be surprised to see just how meager the return of the stock markets are over time. In this period, from 2001 to 2018, the S&P 500 returned less than 6% per year, even with dividends included and even with the last ten years of bull market included. And that was at a peak drawdown of over half.

Another point that may surprise some is the level of the Sharpe ratios. None are over 1. There is an unfortunate misconception that a Sharpe of less than one is poor. That's not necessarily so. In fact, for systematic strategies it's unusual to see realized Sharpe ratios of over one.

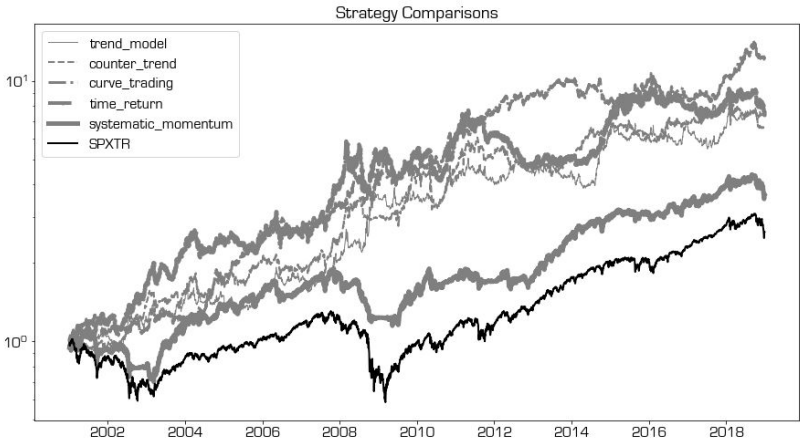


Figure 19-1 Comparing Futures Models

Figure 19-1 shows the long term development of these five strategies, compared to that of the stock market. On such a long time scale, the index comparison hardly seems fair. But the fact is that in the shorter run, you will always be compared to it. This is the curse of the business.

Remember that the reason that these backtests start in 2001 is that a current, and hopefully soon addressed issue in Zipline makes it tricky to use pre-2000 data. The fact that the equity index starts off with a nose dive might make this comparison a little unfair, and for that reason, I will also show you the same graph starting in 2003, the bottom

of the bear market. I won't do one from the bottom of the 2008-2009 bear market. That would just be plain silly. Comparing perfect market timing into the longest lasting bull market of a generation with alternative strategies does not make any sense.

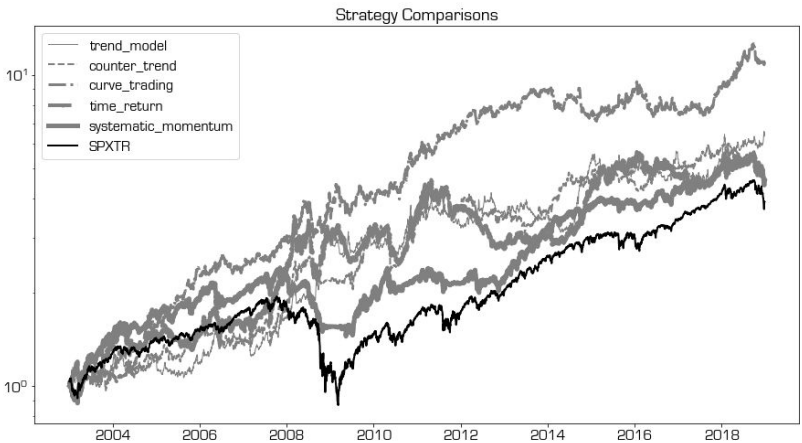


Figure 19-2 Comparison, starting from 2003

Even if we would have had the foresight of buying the index with impeccable timing at the bottom of the tech crash, the index would still have shown lower return and deeper drawdowns.

Combining the

Models

Everyone knows that diversification is beneficial. At least everyone should know that. But most people think of diversification only in terms of holding multiple positions. That's all fine and well, but you can also find added value in diversifying trading styles. Think of a single trading model as a portfolio component.

What you may find is that an overall portfolio of models can perform significantly better than any of the individual strategies that goes into it. I will demonstrate this with a simple portfolio, consisting of the five trading models we have seen so far.

As we have five models, we will allocate an equal weight of 20% of our capital to each. The rebalance period is monthly, meaning that we would need to adjust all positions accordingly each month, resetting the weight to the target 20%. Such a rebalance frequency on a model level can be both difficult and time

consuming for smaller accounts, but is perfectly reasonable on a larger scale. Feel free to repeat this experiment with yearly data if you like. Making portfolio calculations like this is an area where Python shines compared to other languages.

You can read about how this portfolio combination was calculated and see the code for it in chapter 20.

Table 19.2 Portfolio of Futures Models

trend	0.98
counter_trend	-30.09%
curve	1.22
time_return	-20.78%
syste	0.76
Combined	-14.83%

Table 19.2 shows a comparison of the performance of each individual model, as well as the overall stock market, to that of the combined portfolio. These numbers should be abundantly clear. The combined portfolio far outperformed each individual strategy, at lower volatility. We got a

higher annualized return, a lower maximum drawdown, lower volatility, higher Sharpe, etc.

I hope this will help clarify my insistence on that you need to look at the detailed return profile when evaluating a new trading model. It's not necessarily the return per se that you are after, but rather the profile of it, and how well it fits your existing models.

You may find a model with a low expected return over time, but which also has a low or negative correlation to other models, and thereby can greatly help your overall, combined portfolio of trading models.

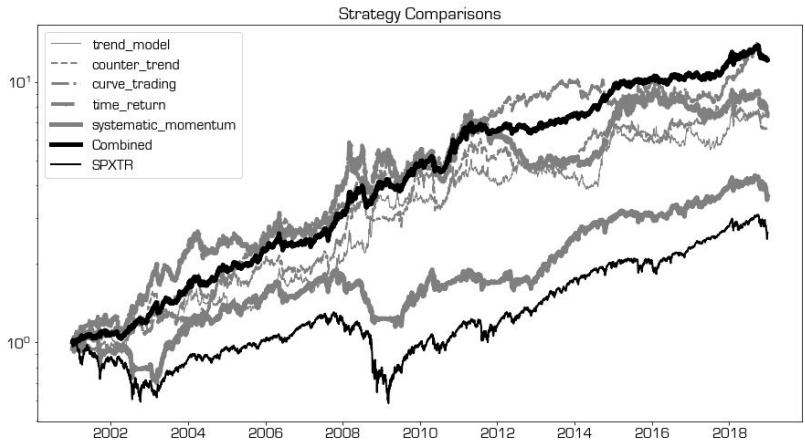


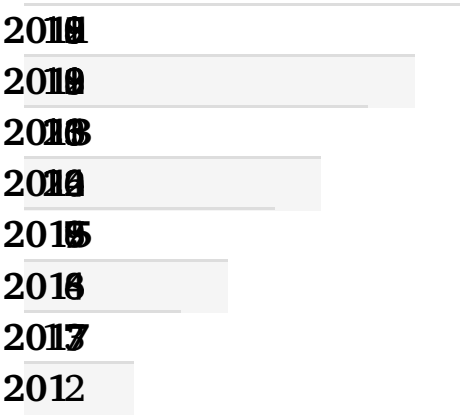
Figure 19-3 Portfolio of Trading Models

You will also see in Figure 19-3 as well as Table 19.3 that the overall return profile seems significantly more attractive, once the models are combined. As the individual models often have their gains and losses as different times from each other, they complement each other well and help smooth out long term volatility. The drawdowns become subdued, resulting in a higher long term return.

While it was a close call some years, in the end, not a single year of this combined portfolio ended up losing money.

Table 19.3 Holding Period Analysis for Combined Model

Yea											
20											
20											
20											
20											
20											
20											
20											
20											
20											
20											



Implementing a Portfolio of Models

While a demonstration like this may seem to show a simple solution to all your investment vows, implementation may not be as easy. Each of these models require millions to trade. Clearly, trading them all requires even more millions. I recognize that not every reader of this book has a spare hundred million laying around to be traded. But even if you are one of those unfortunate few non-billionaires out there, understanding the power of combining different approaches can be greatly helpful.

Insufficient funds is not the only potential problem with building portfolios of models. It can be highly complex in practice to implement the model combination shown in this chapter. As the complexity rises, you lack the simple overview which is possible with a single model, and you may need more sophisticated software to keep track of positions, signals, risk allocation etc.

A professional trading organization can build the capability of trading such complex combinations, to monitor the risk and build proper reporting and analysis. For individual traders, this may not be a possibility.

Of course, there is another way to look at it. An understanding of how complex portfolios of models can be constructed and implemented can help you acquire the skillset needed to land a good job in the industry. Working at the sharp end of the industry has the potential to earn you far more money than you could possibly make trading your own portfolio.

Never forget that the interesting money in this business is made from trading other people's money. Whether or not you personally have the money to trade such models is not the important part. Well, it would be nice to have that of course. But you can still make use of this kind of knowledge and you can still profit from it. If you land a good job with a hedge fund or similar, you will probably get paid far more by working for them than you could by trading your own money anyhow.

20 Performance Visualization and Combinations

In the previous sections you have seen various tables and graphs, showing the performance of backtests. This being a fully transparent book and all, I will show you just how those visualizations were made. These are really simple things to

get done in Python. All we need is a time series to work with, and we can build all kinds of graphs, analytics, tables and other outputs.

Storing Model Results

In chapter 8 we looked at some ways of extracting results from a Zipline backtest and constructing some analytics and charts based on that. The same principles apply here, but the difference is that it would be helpful to store the backtests for further analysis later on.

As you have seen by now, when we run a Zipline backtest we get the results returned to us. In the sample models earlier in this book, this tends to look something like this.

```
perf = zipline.run_algorithm(
    start=start, end=end,
    initialize=initialize,
    analyze=analyze,
    capital_base=millions_traded * 1000000,
    data_frequency = 'daily',
    bundle='futures' )
```

In this case, after the backtest run is

completed, the variable perf will hold all the results. This is a **DataFrame** , consisting of quite a lot of different data which was collected or calculated during the run. If you are looking to store just the portfolio value for each day in the backtest, you can do that very easily. This is stored in the column portfolio_value and that means that we can simply save it as a comma separated file in this one row of code.

```
perf.portfolio_value.to_csv('model_performance.csv')
```

That's all that it takes, and that one line will save the portfolio value, often called the equity curve, of the backtest to a file with the specified name. If you are looking to analyze other aspects of the backtests, you can save the trades done and any other data you will find in perf .

This trick, saving a **DataFrame** to a comma separated file, can be very useful in many instances. Another somewhat related tool that may be particularly helpful at times during debugging, is .to_clipboard( ) . Instead of saving to disk, this will place the **DataFrame** in memory,

in the clipboard. It will be in the correct format so that you can paste it straight into Excel, if that's your intention. Being able to quickly copy something out to Excel for visual inspection can be quite helpful when debugging.

How the Model Performance Analysis was done

To calculate and visualize the performance analytics for each of the model chapters, I started off with a Jupyter Notebook. As discussed earlier in the book, it makes sense to separate different pieces of logic into different cells in the Notebook. Lower cells can access data that you fetched or calculated in the higher level ones.

After running the backtests, I saved the historical portfolio value to a local CSV file. The performance analysis notebook then reads one of these CSV

files, as well as the benchmark data for the S&P 500 Total Return Index for comparison. This is what the first cell of my notebook does.

```
%matplotlib inline

import matplotlib.pyplot as plt
import pandas as pd

# Where the data is
path = 'data/'

# Set benchmark to compare with
bm = 'SPXTR'
bm_name = 'S&P 500 Total Return'

# These are the saved performance csv files from our
book models.
strat_names = {
    "trend_model" : "Core Trend Strategy",
    "time_return" : "Time Return Strategy",
    "counter_trend" : "Counter Trend Strategy",
    "curve_trading" : "Curve Trading Strategy",
    "systematic_momentum" : "Equity Momentum
Strategy",
}

# Pick one to analyze
strat = 'curve_trading'

# Look up the name
strat_name = strat_names[strat]
```

```
# Read the strategy
df = pd.read_csv(path + strat + '.csv', index_col=0,
parse_dates=True, names=[strat] )

# Read the benchmark
df[bm_name] = pd.read_csv(bm + '.csv', index_col=0,
parse_dates=[0] )

# Limit history to end of 2018 for the book
df = df.loc[:'2018-12-31']

# Print confirmation that all's done
print("Fetched: {}".format(strat_name))
```

After that is done, we have the data that we need in a neatly organized **DataFrame** . The next thing I wanted to do was to make a nice looking table of monthly returns. The Python code for aggregating performance on monthly, as well as yearly frequencies does not take much work. As so often is the case, someone else has already written code for this, and there is no need to reinvent the wheel. In this case, I will use the library **Empyrical** , as that should already be installed on your computer if you followed previous chapters. It comes bundled with the **PyFolio** library that we used earlier in chapter 8.

Therefore, calculating monthly and yearly returns takes only one line of code each. The rest of the following cell is about constructing a neat formatted table. For that, I settled on using good old HTML, just to make sure it looks the way I want to for this book. If you simply want to dump the monthly values in text, most of the next cell is redundant. Most of the following code is just about formatting an HTML table for pretty display.

```
# Used for performance calculations
import empyrical as em

# Used for displaying HTML formatted content in
notebook
from IPython.core.display import display, HTML

# Use Empyrical to aggregate on monthly and yearly
periods
monthly_data =
em.aggregate_returns(df[strat].pct_change(),'monthly')
yearly_data =
em.aggregate_returns(df[strat].pct_change(),'yearly')

# Start off an HTML table for display
table = """
<table id='monthlyTable' class='table table-hover
table-condensed table-striped'>
<thead>
<tr>
```

```
<th style="text-align:right">Year</th>
<th style="text-align:right">Jan</th>
<th style="text-align:right">Feb</th>
<th style="text-align:right">Mar</th>
<th style="text-align:right">Apr</th>
<th style="text-align:right">May</th>
<th style="text-align:right">Jun</th>
<th style="text-align:right">Jul</th>
<th style="text-align:right">Aug</th>
<th style="text-align:right">Sep</th>
<th style="text-align:right">Oct</th>
<th style="text-align:right">Nov</th>
<th style="text-align:right">Dec</th>
<th style="text-align:right">Year</th>
</tr>
</thead>
<tbody>
<tr>"""

first_year = True
first_month = True
yr = 0
mnth = 0

# Look month by month and add to the HTML table
for m, val in monthly_data.iteritems():
    yr = m[0]
    mnth = m[1]

    # If first month of year, add year label to table.
    if(first_month):
        table += "<td align='right'><b>{}</b></
td>\n".format(yr)
        first_month = False
```

```
    # pad empty months for first year if sim doesn't
start in January
    if(first_year):
        first_year = False
        if(mnth > 1):
            for i in range(1, mnth):
                table += "<td align='right'>-</td>\n"
```

```
    # Add the monthly performance
    table += "<td align='right'>{:+.1f}</td>
\n".format(val * 100)

    # Check for December, add yearly number
    if(mnth==12):
        table += "<td align='right'><b>{:+.1f}</
b></td>\n".format(yearly_data[yr] * 100)
        table += '</tr>\n <tr> \n'
        first_month = True

# add padding for empty months and last year's value
if(mnth != 12):
    for i in range(mnth+1, 13):
        table += "<td align='right'>-</td>\n"
        if(i==12):
            table += "<td align='right'><b>{:+.1f}
</b></td>\n".format(
                yearly_data[yr] * 100
            )
            table += '</tr>\n <tr>
\n'

# Finalize table
table += '</tr>\n </tbody> \n </table>'
```

```
# And display it.
display(HTML(table))
```

That should output a table looking just like the ones you have seen a few times in this book, for each of the strategy analysis chapters. Next there is the performance chart. In the earlier sections, I used a logarithmic chart to compare each strategy to the equity index. It may not make much sense to compare your strategy to the S&P 500, but it's highly likely that others will compare you to it, whether it's logical or not.

In the same chart image, I also had a drawdown plot as well as a 6 months rolling correlation. These are very easy to calculate, and you should already know how to make the graphs. There is also some code in there to make the next really big and to get the lines to be black and grey for the book.

```
import matplotlib

# Assumed trading days in a year
yr_periods = 252

# Format for book display
```

```
font = {'family' : 'eurostile',
        'weight' : 'normal',
        'size'   : 16}
matplotlib.rc('font', **font)


# Rebase to first row with a single line of code
df = df / df.iloc[0]

# Calculate correlation
df['Correlation'] =
df[strat].pct_change().rolling(window=int(yr_periods /
2)).corr(df[bm_name].pct_change())

# Calculate cumulative drawdown
df['Drawdown'] = (df[strat] / df[strat].cummax()) - 1

# Make sure no NA values are in there
df.fillna(0, inplace=True)

# Start a plot figure
fig = plt.figure(figsize=(15, 12))

# First chart
ax = fig.add_subplot(311)
ax.set_title('Strategy Comparisons')
ax.semilogy(df[strat], '-',label=strat_name,
color='black')
ax.semilogy(df[bm_name] , '--', color='grey')
ax.legend()

# Second chart
ax = fig.add_subplot(312)
ax.fill_between(df.index, df['Drawdown'],
```

```
label='Drawdown', color='black')
ax.legend()

# Third chart
ax = fig.add_subplot(313)
ax.fill_between(df.index,df['Correlation'], label='6M
Rolling Correlation', color='grey')
ax.legend()
```

Finally, for each chapter I did a so called holding period table, that shows your percentage return if you started it in January of a given year, and held for a certain number of full years. Again I elected to use HTML output to ensure that it can be displayed nicely for this book. Since the width of these pages is limited, I also rounded the numbers down to full percent.

```
def holding_period_map(df):
    # Aggregate yearly returns
    yr = em.aggregate_returns(df[strat].pct_change(),
'yearly')

    yr_start = 0
```

```
    #Start off table
    table = "<table class='table table-hover table-
condensed table-striped'>"
    table += "<tr><th>Years</th>"
```

```
    # Build the first row of the table
    for i in range(len(yr)):
        table += "<th>{}</th>".format(i+1)
    table += "</tr>"
```

```
    # Iterate years
    for the_year, value in yr.iteritems():
        # New table row
        table += "<tr><th>{}</
th>".format(the_year)
```

```
        # Iterate years held
        for yrs_held in (range(1, len(yr)+1)): # Iterates
yrs held
            if yrs_held   <= len(yr[yr_start:yr_start +
yrs_held]):
                ret = em.annual_return(yr[yr_start:yr_start
+ yrs_held], 'yearly' )
                table += "<td>{:+.0f}</
td>".format(ret * 100)
        table += "</tr>"
        yr_start+=1
    return table

table = holding_period_map(df)
display(HTML(table))
```

How the Combined Portfolio Analysis

was done

In chapter 19 we looked at the diversification benefits of combining models and trading portfolios of models. The method used in that chapter was to rebalance at the start of every month, resetting the weight of each strategy at that interval. In the code shown here, I will also give you another method of rebalancing. As you will see in the next code segment, you can also trigger the rebalance on percentage divergence, if the market developments have pushed any strategy to be more than a certain percent off.

This is a somewhat advanced topic, and really goes beyond what I was planning to show in this book. I include this source code here anyhow, in the interest of being transparent. I will however avoid the potentially lengthy discussion of how this code is constructed and the reason for it. It uses some tricks to enhance performance, making use of **Numpy** to speed things up.

Once you feel comfortable with Python and backtesting, this is a topic you may want to dig deeper into. How to use optimize complex operations and speed code up. But it's out of scope for this book.

```
import pandas as pd
import numpy as np

base_path = '../Backtests/'
# Rebalance on percent divergence
class PercentRebalance(object):
    def __init__(self, percent_target):
        self.rebalance_count = 0
        self.percent_target = percent_target
```

```
    def rebalance(self, row, weights, date):
        total = row.sum()
        rebalanced = row
        rebalanced = np.multiply(total, weights)
        if np.any(np.abs((row-rebalanced)/rebalanced) >
(self.percent_target/100.0)):
            self.rebalance_count = self.rebalance_count +
1
            return rebalanced
        else:
            return row

# Rebalance on calendar
class MonthRebalance(object):
    def __init__(self, months):
        self.month_to_rebalance = months
```

```
        self.rebalance_count = 0
        self.last_rebalance_month = 0

    def rebalance(self, row, weights, date):
        current_month = date.month

        if self.last_rebalance_month != current_month:
            total = row.sum()
            rebalanced = np.multiply(weights, total)
            self.rebalance_count = self.rebalance_count +
1
            self.last_rebalance_month = date.month
            return rebalanced
        else:
            return row
# Calculate the rebalanced combination
def calc_rebalanced_returns(returns, rebalancer,
weights):
    returns = returns.copy() + 1
```

```
    # create a numpy ndarray to hold the cumulative
returns
    cumulative = np.zeros(returns.shape)
    cumulative[0] = np.array(weights)

    # also convert returns to an ndarray for faster
access
    rets = returns.values

    # using ndarrays all of the multiplicaion is now
handled by numpy
    for i in range(1, len(cumulative) ):
        np.multiply(cumulative[i-1], rets[i],
out=cumulative[i])
```

```
        cumulative[i] =
rebalancer.rebalance(cumulative[i], weights,
returns.index[i])

    # convert the cumulative returns back into a
dataframe
    cumulativeDF = pd.DataFrame(cumulative,
index=returns.index, columns=returns.columns)

    # finding out how many times rebalancing happens
is an interesting exercise
    print ("Rebalanced {}
times".format(rebalancer.rebalance_count))

    # turn the cumulative values back into daily returns
    rr = cumulativeDF.pct_change() + 1
    rebalanced_return = rr.dot(weights) - 1
    return rebalanced_return

def get_strat(strat):
    df = pd.read_csv(base_path + strat + '.csv',
index_col=0, parse_dates=True, names=[strat] )
    return df
# Use monthly rebalancer, one month interval
rebalancer = MonthRebalance(1)

# Define strategies and weights
portfolio = {
    'trend_model': 0.2,
    'counter_trend': 0.2,
    'curve_trading': 0.2,
    'time_return': 0.2,
    'systematic_momentum' : 0.2,
}
```

```
# Read all the files into one DataFrame
df = pd.concat(
        [
            pd.read_csv('{}{}.csv'.format(
                        base_path,
                        strat
                        ),
                        index_col=0,
                        parse_dates=True,
                        names=[strat]
                       ).pct_change().dropna()
            for strat in list(portfolio.keys())
        ], axis=1
)

# Calculate the combined portfolio
df['Combined'] = calc_rebalanced_returns(
    df,
    rebalancer,
    weights=list(portfolio.values())
    )

df.dropna(inplace=True)
# Make Graph
import matplotlib
import matplotlib.pyplot as plt

include_combined = True
include_benchmark = True
benchmark = 'SPXTR'

if include_benchmark:
    returns[benchmark] =
```

```
get_strat(benchmark).pct_change()

#returns = returns['2003-1-1':]
normalized = (returns+1).cumprod()

font = {'family' : 'eurostile',
        'weight' : 'normal',
        'size'   : 16}

matplotlib.rc('font', **font)


fig = plt.figure(figsize=(15, 8))

# First chart
ax = fig.add_subplot(111)
ax.set_title('Strategy Comparisons')

dashstyles = ['-','--','-.',':.-', '-']
i = 0
for strat in normalized:
    if strat == 'Combined':
        if not include_combined:
            continue
        clr = 'black'
        dash = '-'
        width = 5
    elif strat == benchmark:
        if not include_benchmark:
            continue
        clr = 'black'
        dash = '-'
        width = 2
    #elif strat == 'equity_momentum':
```

```
    #    continue

    else:
        clr = 'grey'
        dash = dashstyles[i]
        width = i + 1
        i += 1
    ax.semilogy(normalized[strat], dash, label=strat,
color=clr, linewidth=width)

    ax.legend()
```

21 You can't beat all of the Monkeys all of the Time

You shouldn't worry if you are being outperformed by a primate. It happens to the best of us. But let's take this from the beginning.

Back in 1973 the iconic book *A Random Walk Down Wall Street* was published, written by Burton Malkiel.

There is much in this book which has not stood the test of time very well but there are also some parts which have been quite prophetic. The irony of it is of course that while much of his analysis was spot on, the conclusions turned out a little different.

The book is the main proponent of the so called Efficient Market Hypothesis. The academic theory that states that asset prices fully reflect all known information and that it's impossible to beat the market over time. Movement in stock prices when no new information has been made available is purely random and thereby unpredictable.

This theory has of course as much value as an academic paper proving that bumble bees are unable to fly. Empirical observation has continued to barrage this academic model for decades, and only career academics take it seriously anymore.

But I'm not writing this to discredit professor Malkiel's theories. After all, Warren Buffett has already done a pretty

good job on that front. No, I'm bringing this book up because of a famous quote. One which turned out to be truer than the author likely expected, or desired.

"A blindfolded monkey throwing darts at a newspaper's financial pages could select a portfolio that would do just as well as one carefully selected by experts."

Now we are talking. This is absolutely true. The mistake is in marginalizing the monkeys and discrediting their abilities. And of course, I fail to see the benefits of blindfolding the poor monkey, other than making him look funny.

The ability of primates to outperform the stock market is undisputed. The question is what we do with this knowledge. Professor Malkiel's conclusion was that we should abandon all hope of competing with the furry little guys and simply buy mutual funds. That may have seemed like a good idea back in the 70s, but that was before we had

conclusive evidence of the failure of the mutual fund industry. The chimps will stand a better chance at delivering performance than a mutual fund.

Pointing out that mutual funds are a bad idea is not exactly bringing anything new to the table. The quarterly S&P Indices versus Active (SPIVA, 2019) reports published by S&P Dow Jones Indices are devastating. Around 80% of all mutual funds fail to match their benchmark index in any given 3 or 5 year period. The ones that do beat their benchmark tend to be different each year, indicating the impact of pure chance.

Mutual funds are designed to generate maximum revenue for fund managers and banks while ensuring that they underperform by a small enough margin that most retail savers don't realize what is going on. It's a nice business model. You launch a fund that invests almost all the assets like a given index composition. Small deviations are allowed, but within very strict tracking error budgets. The design is to avoid any noticeable deviation from the index.

Mutual funds charge a fixed yearly management fee, as a percentage of the assets, but that's of course not the entire revenue base. Most banks that offer mutual funds also have brokerage departments, and this is of course where the fund trades. The flow information is valuable, knowing when and how the funds will rebalance and what orders they place. There are plenty of ways to make good money from mutual funds and all you need to do is to ensure that they won't deviate much from the benchmark index.

Back in the 70s when the book in question was written, this pattern was probably not as visible. Mutual funds must have seemed like a really good idea back then. What we know now is different, and it's highly unlikely for a mutual fund to outperform the proverbial primate. The mutual funds are not even trying to do that. There is no profitability in such an endeavor, not compared to playing it safe, showing a small underperformance and cashing in on big fees.

But I digress. None of this is new

information. After all, pointing out that mutual funds are not great investments is at this point far beyond beating a dead horse. It more resembles yelling at the can of dog food in the supermarket where the horse eventually ended up. Buying mutual funds is not the answer.

Perhaps we should just resign and hire an office chimp. Yes, I'm aware that Malkiel called for a monkey, but using an ape barely seems like cheating. There are after all clear advantages of chimps compared to recruiting from the hedge fund industry, such as lower salary, better behavior and calmer tempers. Not everyone is however lucky enough to be able to retain the services of a cooperative primate. The good news is that we have a way of simulating the skills of these remarkable creatures. Random number generators.

Mr. Bubbles goes to Wall Street

Before you set out to design trading strategies for the stock markets, you need to be aware of what your average, reasonably well behaved chimp would be expected to achieve. The fact of the matter is that a random stock picking approach will make a nice profit over time. I'm of course not suggesting that you pick your stocks at random, but this knowledge does have implications.

Perhaps you design a stock picking model that shows quite nice long term results. You have been working on it for a long time, constructing a complex method of combining technical indicators, fundamental ratios and designing exact rules. Now your simulations clearly show that you will make money with these rules.

The question is of course, whether or not you will make more or less money than a random approach, and at higher or lower risk. It's quite easy to construct a model which performs more or less the same as a random approach, and in that case you need to ask yourself if you really added any value.

To demonstrate the principle, I will let our imaginary office chimp, Mr. Bubbles show some of his best strategies.

We will start with a simple classic. The Equal Random 50. The rules are really simple. At the start of every month, Mr. Bubbles will throw imaginary darts at an equally imaginary list of stocks, picking 50 stocks from those that were part of the S&P 500 index on the day of selection. The entire portfolio will be liquidated and replaced with 50 new stocks at the first of every month.

Now don't misunderstand what we are doing here. Only a chimp would actually trade this way. We are doing this to learn about the behavior of stocks. Put that keyboard away. You will have plenty of time to send me angry mails later.

Having selected 50 random stocks, we simply buy an equal amount of each. Not number of shares of course, but in value. In this case, we start off with 100,000 dollars and we invest 2,000 into each stock. This is why we call the strategy Equal Random 50. Equal sizing,

random selection, 50 positions.

When dividends come in, they will be left in cash on the account until the next monthly rebalancing, when it will be used to buy stocks again. This strategy is always fully invested, with the exception of these cash dividends.

This monkey strategy does not care at all about what stocks it's buying or what the overall market is doing. How do you think such a strategy would perform?

The intuitive response, which would be as reasonable as it would be incorrect, is that the results would be centered around the index. Much like how throwing darts at a board. If you aim for the center and throw enough number of times, you should in theory get a fairly even distribution around the center. Unless you have some sort of weird arm that pulls to the left of course.

The logical error with that response is to assume that the index is some sort of average. It's not. The index is a completely different systematic trading strategy. And a poorly designed one at

that.

Since the simulations here use random numbers, it would of course be pointless to do it only once. After all, if you spin the roulette table once or twice, anything can happen. You might even win money. But if you spin it enough times, the end result is highly predictable and there is virtually no chance of you walking away from the Bellagio with any cash.

For every random strategy in this chapter, I will show 50 iterations. A few readers who studied statistics are probably shouting at the pages at this point, but no need to worry. I have done the same simulations with 500 iterations each and the results won't differ enough to warrant the visual mess of showing you 500 runs.

Figure 21-1 shows you the result of these 50 iterations. There is no need to squint to try to make out the differences between all the lines. The most important to read from this chart is where the index is compared to the others. The index, in this case the S&P 500 Total Return Index,

is shown as a thick black line to distinguish it from the others.

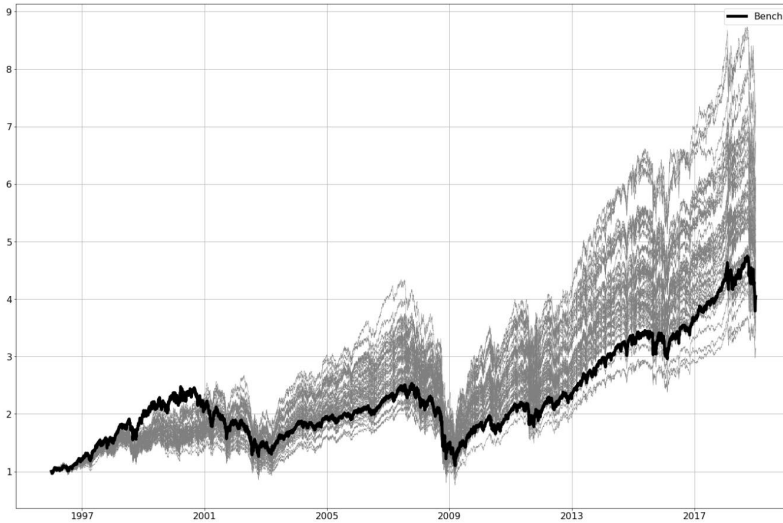


Figure 21-1 Random Stock Picking

The first and most obvious thing you should notice in this picture is that almost all of the random stock selection simulations ended up with more money than the index.

The second thing you might notice is that the random stock picking approach started beating the index just by the turn of the century. Back in the 90's, the index was in the lead. We will return to the reason for this later on.

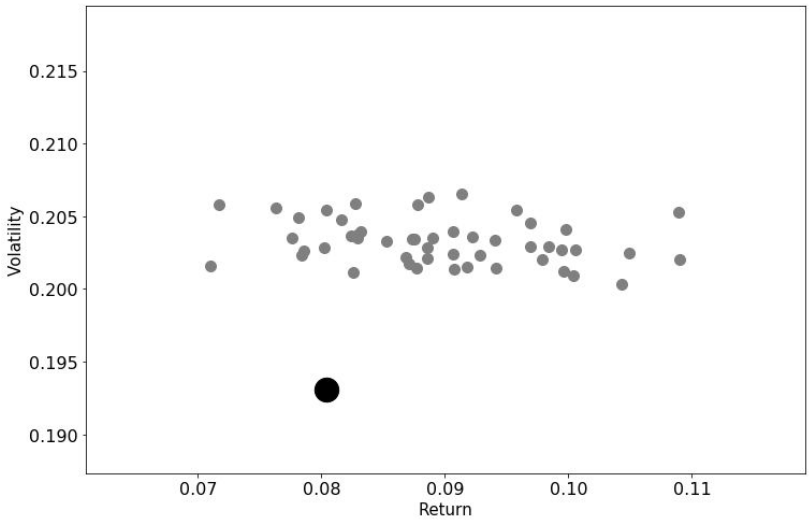


Figure 21-2 Random 50 Returns vs Volatility

The higher overall returns are not free of course. Returns rarely are. Figure 21-2 shows a scatter plot of returns versus volatility. This paints a somewhat different picture. As you can see here, the volatility is higher for all of the random strategies. For some of the random runs, we have about equal returns in the long run at higher volatility, and that's clearly not very attractive, but for many of them the returns are high enough to make up for the added volatility.

You might notice that I haven't mentioned trading costs yet. What about

slippage and commissions? So far, I haven't applied any. No, no, it's not for the reason that you think. It's not to make the monkey selection look better. At least that's not the main reason. The reason is actually that in this case, it wouldn't make a fair comparison if we included these costs.

We are not trying to devise a practical trading plan here, making realistic accounting for costs, slippage and such to get to an expected return number. We are letting a chimp throw darts at a board and there is very little realism about that. No, here we are comparing the concept of the index to the concept of Mr. Bubble's dart throwing skills. If we put trading costs on his strategy, we need to put trading costs on the S&P index as well.

The index does not include any such fees. The index is not constructed as some sort of realistic trading strategy with slippage and commissions. They assume that you can instantly buy and sell at the market at any amount without any fees or slippage at all. So it's only fair that Mr. Bubbles gets to do the same.

Other models in this book use realistic trading costs, but in this more theoretical chapter we will trade for free. For now, I'm just trying to make a point about the stock markets.

Clearly this seemingly ludicrous strategy is outperforming the market in the long run. Well, at least it has a higher than average probability of ending up with higher returns, at slightly higher volatility. At the very least, we can conclude that in comparison to the index methodology, the results are not completely outrageous.

There is no trick at work here. A random stock picking model would have had a high probability of beating the index in terms of returns for the past few decades. There is a reason for this of course, but we are going to throw a few more darts before getting to the actual point of this chapter.

The model we just tried used equal sizing. Perhaps that is the trick here. Just in case, we will remove that factor from the equation. Enter the Random Random

50 Model.

The Random Random 50 is mostly the same as the Equal Random 50. It replaces the entire portfolio monthly based on random selections from the index members. The only difference is that it does not allocate equal weight to the positions. Instead, it allocates a random weight. We always use 100% of available cash, but how much we buy of each stock is completely random.

This does make sense. After all, why would a chimp buy an equal amount of each stock?

What you see here, in Figure 21-3 is a very similar pattern. We again see underperformance in the late 1990's followed by a very strong outperformance during a decade and a half after that.

There is quite a wide span between the best and the worst random iteration here, but even with the drag in the 90's, almost every one beats the index. Keep in mind how this graph would look if you instead plot actual, real life mutual funds against the index in the same way. Out of

50 randomly selected mutual funds, one or two would statistically be above the index. Not more. The very opposite of what the chimp strategies show. Invest with a mutual fund and you have a 20% chance of beating the benchmark. Invest with a chimp and you have an 80% chance.

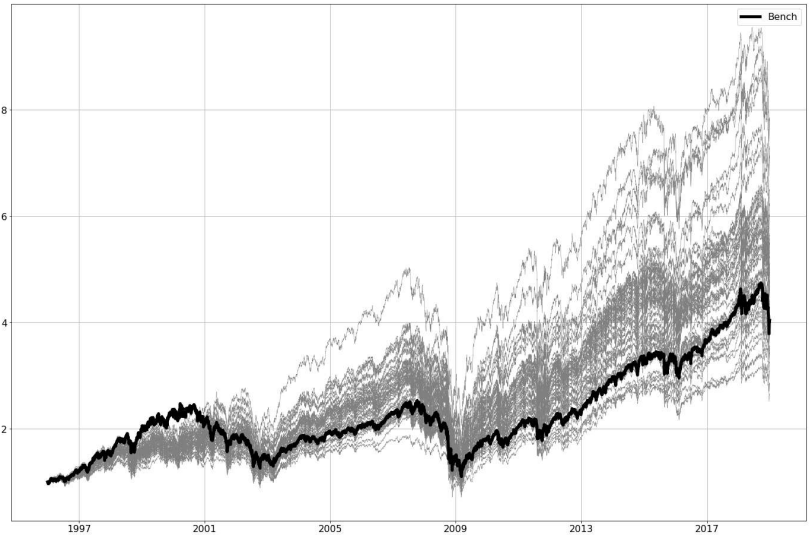


Figure 21-3 Random Selection, Random Size

As for the scatter plot, it again looks quite similar. We have a wider span, both in terms of volatility and returns. That would be expected, given the random position sizes and the extreme allocation effects that might bring.

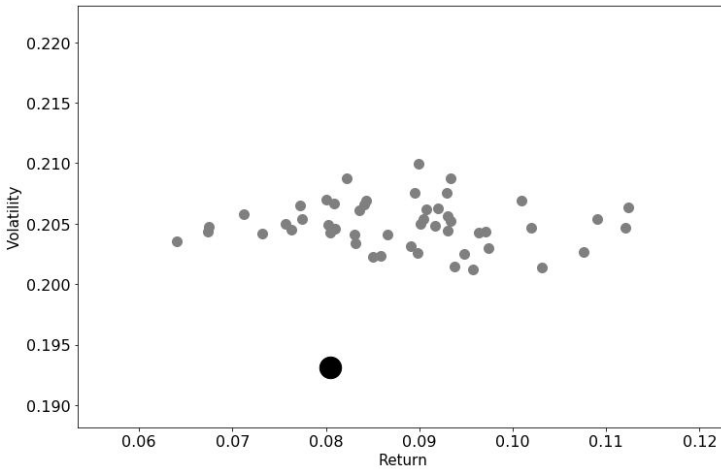


Figure 21-4 Random Selection, Random Size, Returns versus Volatility

Again we have some iterations which are clearly worse than the index. They show higher volatility for lower returns, or similar returns for higher volatility. But most iterations show stronger returns in the long run.

We could take this little game even further and add a randomizer for the number of stocks to pick as well. But it really wouldn't matter all that much. The end result would be predictable. We would see an even wider span in both returns and volatility, and we would still see a higher expected return for the random model.

No, I think we will just skip ahead to the point of this chapter.

The Problem is with the Index

Ok, so now we know that random stock picking outperforms the index, at least in that it has a very high chance of showing superior long term returns. The effect is there with equal sizing, with random sizing and even with random number of stocks. As long as we have a reasonable amount of stocks and avoid going utterly crazy with the allocation, we seem to outperform. At least after the 1990's. So what is the deal here? Did I just waste your time reading about nonsense strategies, or is there a point to be made here?

The issue, which is very easy to overlook, is what we are comparing to. Is there really a good reason to use the index as a benchmark? To understand why random stock selection appears favorable,

you need to understand what the index really is. Just another trading model.

A market index is not some objective measurement of the financial markets. I'm not sure if such a thing exists or even how to go about creating a fair market measurement. In fact, I have never seen an index methodology that's not flawed in one way or another. That's not the fault of the index provider though. No matter how you construct an index, it will by necessity be geared towards some factor or another.

Most modern market indexes are based two main market factors. Long term momentum and market capitalization. Let's first look at why I claim that these indexes are momentum based, and then we will get to the more obvious market cap issue.

What does it take for a stock to join the S&P 500 Index? The actual selection is done on a discretionary basis by a committee, but any stock to be considered needs to fulfill certain criteria. The most important of these are that the stock need

to have a total market worth in excess of 5 billion dollars. There are also rules regarding the liquidity and free float, to ensure that the stock can actually be traded in large scale.

Companies with a value of 5 billion dollars don't just pop into existence. They started out small and grew larger. As the stock price rose, they became more valuable. At some point, they became valuable enough to be included in some index. So in a very real sense, the stocks in the S&P 500 are included primarily because they had strong price performance in the past. This is what makes the index a sort of momentum strategy. Perhaps not a great momentum strategy, but it's nevertheless a major factor in stock selection.

The second factor is closely related. The current market capitalization. However in this case, the issue is in the weighting. Most major indexes are weighted based on market cap. The most obvious exception being the Dow Jones, which is weighted purely on price level.

The issue with market cap is significant. If you were looking for the trick that makes the chimp strategies above appear to work, this is it. The market cap weighting model of most major indexes has the unintended side effect of destroying diversification.

Most people would assume that a broad index like the S&P 500 is highly diversified. After all, it covers the 500 largest companies in the United States. The problem is that the index price development would look almost identical if you simply skip the bottom 300 stocks. This is not a diversified index. It's extremely concentrated in a few massive companies.

The top three largest companies in the index have a combined weight of about 10.5%. That's about the same weight as the bottom 230 stocks. The highest weight company has at this time a weight of almost 4%, while the lowest weight is less than 0.01%. That's not diversification.

What it means is that when you

invest in a traditional index way, you are really buying a factor strategy based on market cap. You are putting all of your money in a small number of extremely large companies. The rest of the stocks in the index really don't matter.

This is why the chimps are outsmarting us. It's not that they are doing something particularly right. It's more like the index is doing something wrong.

At times the large cap stocks outperform the small caps. That happened in the mid to late 1990's. Most of the time however, the large caps underperform the mid and small caps. This effect can be seen even within an index like the S&P 500, where all stocks are by definition large caps. There is quite a large difference between a stock worth a few billions and one large enough to make the GDP of Switzerland look like a rounding error.

The largest companies in the world became that big because they were well performing mid to large caps. But once

you have a market cap equivalent to the GDP of South America, the question is how many times you can possibly double from there. By necessity, these giants become slow movers after a while. They will have much more downside potential than upside.

These highly concentrated indexes are very popular though. One reason is of course that many people don't fully understand how concentrated they are, but that's just a small part of the story. The larger factor at work is group think.

There is a very old industry axiom that's very fitting in this context. "Nobody ever got fired for buying IBM". This proverb comes from the days when IBM was one of the largest companies in the world. The equivalent of Apple or Microsoft now. If you buy the proverbial IBM, you are doing what everyone else is doing, and you can't possibly be blamed for that. Most investments are done on behalf of other people. Fund managers and asset managers of various kinds. From that perspective, not getting blamed is a priority. You get paid by fees. If you lose

money when everyone else loses money, you will be fine. Nobody can blame you for that. But if you try to do your best and risk failing, you might get fired. So it's in your interest to buy IBM.

The reason for the seemingly superior performance of the random strategies is that they did not use market capitalization weighting. This is also the reason why they underperformed in the 1990's. During the IT crash of 2000 to 2002, the largest companies took the biggest beating. That's why most random strategies did not.

The point here, I suppose, is that you need to be aware of what the index is. You also need to decide if your goal is to beat the index, or if your goal is index independent.

In the financial industry we talk about relative strategies and absolute return strategies. All too often, these very different approaches get mixed up. A relative strategy must always be compared to its benchmark index, while an absolute return strategy should not.

This can often be a problem in the industry. The fact that most people are not aware of the distinction. If you have an absolute return strategy and end a year at 15 percent up, that would normally be considered a good year. If the equity markets were down 20 percent that year, your clients will trust you to watch their kids over the weekend. But if the stock markets were up 25 percent, you quickly get taken off the Christmas card list.

Finding Mr. Bubbles

In case you would like to try this at home, I will provide you the source code used for this experiment. Much of the code here is the same as the momentum model, but without the fancy momentum and allocation logic. Before showing the entire code, let's look at the interesting parts.

Apart from the same import statements that we had last time, we also need to get a library that can generate random numbers for us.

```
# To generate random numbers
from random import random, seed, randrange
```

We're using this to select random stocks. The actual list of eligible stocks are fetched the same way as before.

```
  # Check eligible stocks
  todays_universe = [
     symbol(ticker) for ticker in

context.index_members.loc[context.index_members.index
< today].iloc[-1,0].split(',')
  ]
```

The actual selection is done by a loop that's run once a month. Here we loop once for each stock that we'd need to select, and use the **random** library to generate a number. We make sure that this number will be between zero and the length of the list of stocks, minus one. Remember that lists are zero based.

You also see the use of `pop( )` here, which is a convenient way of both selecting an item from a list and removing it from the list at the same time. This ensure that we won't pick the same stock twice.

```
  for i in np.arange(1, number_of_stocks +1):
     num = randrange(0, len(todays_universe) -1)
```

```
        buys.append(todays_universe.pop(num))
```

The only other part here that is really new, is how we make a backtest run multiple times with the results stored. In this case, all that I was interested in was the portfolio value over time, so that's all that I stored in this loop.

```
# Run the backtests
for i in np.arange(1, number_of_runs + 1):
    print('Processing run ' + str(i))
```

```
    result = zipline.run_algorithm(
        start=start, end=end,
        initialize=initialize,
        capital_base=100000,
        data_frequency = 'daily',
        bundle='ac_equities_db' )
```

```
    df[i] = result['portfolio_value']
```

In the end, we will have a **DataFrame** with the portfolio value development over time, one column for each backtest run. Now all you need to do is to make some nice graphs or calculate some useful analytics.

The full source for the random stock picking model follows here below.

```
%matplotlib inline

import zipline
from zipline.api import order_target_percent, symbol,
set_commission, \
    set_slippage, schedule_function, date_rules,
time_rules

from zipline.finance.commission import PerTrade,
PerDollar
from zipline.finance.slippage import
VolumeShareSlippage, FixedSlippage

from datetime import datetime
import pytz
import pandas as pd
import numpy as np

# To generate random numbers
from random import random, seed, randrange

"""
Settings
"""
number_of_runs = 2
random_portfolio_size = False
number_of_stocks = 50  # portfolio size, if not
random
sizing_method = 'equal' # equal or random

enable_commission = False
commission_pct = 0.001
enable_slippage = False
slippage_volume_limit = 0.025
```

```
slippage_impact = 0.05

def initialize(context):
    # Fetch and store index membership
    context.index_members = pd.read_csv('../data/
index_members/sp500.csv', index_col=0,
parse_dates=[0])
```

```
    # Set commission and slippage.
    if enable_commission:
        comm_model = PerDollar(cost=commission_pct)
    else:
        comm_model = PerDollar(cost=0.0)
    set_commission(comm_model)
```

```
    if enable_slippage:

slippage_model=VolumeShareSlippage(volume_limit=slippage
price_impact=slippage_impact)
```

```
    else:
        slippage_model=FixedSlippage(spread=0.0)
    set_slippage(slippage_model)
```

```

```

```
    schedule_function(
        func=rebalance,
        date_rule=date_rules.month_start(),
        time_rule=time_rules.market_open()
    )

def rebalance(context, data):
```

```
    today = zipline.api.get_datetime()
```

```
    # Check eligible stocks
    todays_universe = [
        symbol(ticker) for ticker in

context.index_members.loc[context.index_members.index
< today].iloc[-1,0].split(',')
    ]
```

```
    # Make a list of stocks to buy
    buys = []
```

```
    # To modify global variable, and not make new one
    global number_of_stocks
```

```
    # If random stockss selected
    if random_portfolio_size:
        # Buy between 5 and 200 stocks.
        number_of_stocks = randrange(5, 200)
```

```
    # Select stocks
    for i in np.arange(1, number_of_stocks +1):
        num = randrange(0, len(todays_universe) -1)
        buys.append(todays_universe.pop(num))
```

```
    # Sell positions no longer wanted.
    for security in context.portfolio.positions:
        if (security not in buys):
            order_target_percent(security, 0.0)
```

```
    #Make an empty DataFrame to hold target position
sizes
    buy_size = pd.DataFrame(index=buys)
```

```
    # Get random sizes, if enabled.
    if sizing_method == 'random':
        buy_size['rand'] = [randrange(1,100) for x in
buy_size.iterrows()]
        buy_size['target_weight'] = buy_size['rand'] /
buy_size['rand'].sum()
    elif sizing_method == 'equal':
        buy_size['target_weight'] = 1.0 /
number_of_stocks
```

```
    # Send buy orders
    for security in buys:
        order_target_percent(security,
buy_size.loc[security, 'target_weight'])
```

```
start = datetime(1996, 1, 1, tzinfo=pytz.UTC)
end = datetime(2018, 12, 31, tzinfo=pytz.UTC)

# Empty DataFrame to hold the results
df = pd.DataFrame()

# Run the backtests
for i in np.arange(1, number_of_runs + 1):
    print('Processing run ' + str(i))
```

```
    result = zipline.run_algorithm(
        start=start, end=end,
        initialize=initialize,
```

```
        capital_base=100000,
        data_frequency = 'daily',
        bundle='ac_equities_db' )

    df[i] = result['portfolio_value']

print('All Done. Ready to analyze.')
```

After this code is all done, the resulting data is in the `df` object, and you can save it to disk or use techniques described in previous chapter to chart or analyze it.

22 Guest Chapter: Measuring Relative Performance

Robert Carver is an independent systematic futures trader, writer and research consultant; and is currently a visiting lecturer

at Queen Mary, University of London. He is the author of "Systematic Trading: A unique new method for designing trading and investing systems", and "Smart Portfolios: A practical guide to building and maintaining intelligent investment portfolios".

Until 2013 Robert worked for AHL, a large systematic hedge fund, and part of the Man Group. He was responsible for the creation of AHL's fundamental global macro strategy, and then managed the funds multi-billion dollar fixed income portfolio. Prior to that Robert traded exotic derivatives for Barclays investment bank.

Robert has a Bachelors degree in Economics from the University of Manchester, and a Masters degree, also in Economics, from Birkbeck College, University of London.

Python is a great tool for running backtests, and providing plenty of diagnostic information about the potential performance of your trading strategies. But tools are only useful if you know how to use them properly – otherwise they can

be dangerous. Try cutting a piece of wood with a chainsaw whilst blindfolded, and see how well that goes. Similarly, a superficially wonderful backtest can result in serious damage to your wealth, if it turns out that the strategy you have developed is not robust.

In this chapter I'm going to focus on analysing ***relative*** performance: one trading strategy against another. There are several different situations in which relative performance matters. Firstly, all strategies should be compared against some kind of benchmark. For example, if your strategy focuses on buying selected S&P 500 stocks, then it ought to be able to outcompete the S&P 500 index.

Another important comparison is between different variations of the same strategy. Finding the optimal strategy variation is sometimes known as **fitting** . Consider the futures trend following model that Andreas introduced in chapter 21. The entry rule has three parameters: the two moving average lengths, and the number of days we look back for a breakout. Andreas uses values of 40 days,

80 days, and 50 days for these parameters. These are probably not bad parameter values (ignore his jokey exterior: Andreas is no idiot). But perhaps we could do better? To test this out, we could generate backtests for different sets of parameter values, and pick the best one.

Comparisons between strategies can also be important if you are running multiple strategies, and need to decide how much of your valuable cash to allocate to each one. A strategy that does better in backtest is probably worthy of a larger allocation.

The final comparison you might want to make is between an older, and newer, version of your trading strategy. Once you've got a strategy trading there is an almost unbearable urge to tinker and improve it. Before long you will find yourself with a slightly improved backtest. However a proper comparison is required to see if it's worth upgrading to the latest model, or sticking with the trusty original version.

A key issue with all these different comparisons is **significance** . Your strategy should be significantly better than a benchmark. If you're going to choose one variation of a trading strategy, with a particular set of parameters, then it ought to be significantly better than the alternatives. When allocating capital amongst strategies, if you're going to put a lot of your eggs in a given basket, then you need to be confident that basket is significantly better than other available egg containers. Finally, if you're thinking about switching from one strategy to another, you need to determine whether the improvement in returns is significant enough to justify the work.

Another important topic is **comparability** . If you haven't chosen a benchmark that's appropriate for your strategy, then your results will be meaningless. It doesn't make much sense for a diversified futures trend follower to use the S&P 500 index as a benchmark, although you will often see this comparison in marketing materials. The benchmark and the strategy should also

have similar risk, or the riskier strategy will have an advantage.

In this chapter I'm going to focus on comparing the returns from Andreas' Equity Momentum model with a benchmark: the S&P 500 index. However, the general ideas will be applicable in other situations as well.

Some of these kinds of comparisons can be done automatically using tools like pyfolio, but in this chapter I'm going to show you how to do your own calculations from first principles. This will give you a better understanding of what is going on.

First let's get the returns. You can download these files from Andreas' website, along with this source code. The files you need are SPXTR.csv and equity_momentum.csv, both located in the source code download file, under folder Backtests. You'll find the files at **www.followingthetrend.com/trading-evolved** .

```
import pandas as pd
data_path = '../Backtests/'
```

```
strat = "equity_momentum"
bench = 'SPXTR'
benchmark = pd.read_csv("{}{}.csv".format(data_path,
bench), index_col=0, header=None,
parse_dates=True)
strategy = pd.read_csv("{}{}.csv".format(data_path,
strat), index_col=0, header=None, parse_dates=True)
```

When comparing two strategies we need to make sure the returns cover the same time period:

```
first_date = pd.Series([benchmark.index[0],
strategy.index[0]]).max()
benchmark = benchmark[first_date:]
strategy = strategy[first_date:]
```

We also need to make sure that our returns occur at the same time intervals, by matching the benchmark returns to the strategy by using ‘ ffil l ’: forward fill to replace any missing data:

```
benchmark =
benchmark.reindex(strategy.index).ffill()
```

Now we are going to plot the account curves of the two strategies. I prefer to do this using cumulative percentage returns.

The first step is to calculate percentage returns:

```
benchmark_perc_returns = benchmark.diff()/
benchmark.shift(1)
strategy_perc_returns = strategy.diff()/strategy.shift(1)
```

Now we cumulate these:

```
benchmark_cum_returns =
benchmark_perc_returns.cumsum()
strategy _cum_returns =
strategy_perc_returns.cumsum()
both_cum_returns =
pd.concat([benchmark_cum_returns,
strategy_cum_returns], axis=1)
both_cum_returns.columns=['benchmark', 'strategy']
both_cum_returns.plot()
```

Figure 22-1 Cumulative Returns

Incidentally, this is equivalent to plotting the account curves on a logarithmic scale. With this kind of plot it's easier to see performance over the entire history of the account curve, as a

10% profit or loss has exactly the same scale regardless of which time period we are looking at.

On the face of it, it looks like the strategy is significantly better than the benchmark. Let's look at the difference in performance.

```
diff_cum_returns = strategy_cum_returns - benchmark_cum_returns
diff_cum_returns.plot()
```

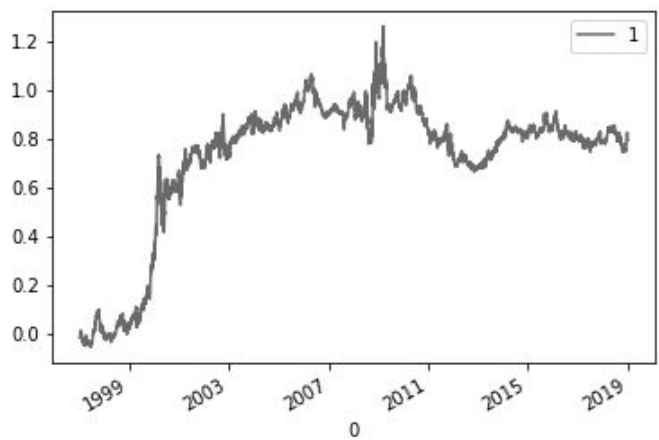


Figure 22-2 Difference in Cumulative Returns

Now we can see more clearly that the outperformance is confined to the first 7 years or so of the backtest. After that the relative performance is flat. Is this change in performance significant? That's

a good question, and we will return to it later when we have the tools in place answer it.

Before we do that, let's start with some basic analysis. How much money does each strategy make? Personally, I like to calculate annual returns. The returns are daily, so we need to multiply by the number of days in each year. Our returns exclude weekends and other non-trading days. There are roughly 256 business days in a year (in fact there is another reason why I use this specific number):

```
print(strategy_perc_returns.mean()*256)
```

0.124743

```
print( benchmark_perc_returns.mean()*256 )
```

0.094309

That's a pretty decent outperformance by the strategy, 12.5% a year versus 9.4% for the benchmark: just over 3%. But it wouldn't be a fair comparison if the strategy was a lot riskier than the benchmark. Let's check, using annual standard deviation of returns as our measure of risk. That's not a perfect

measure of risk, as it assumes our returns follow a particular statistical distribution known as the Gaussian normal. However it will do for now.

```
print(strategy_perc_returns.std()*16)
```

0.189826

```
print(benchmark_perc_returns.std()*16)
```

0.192621

Notice that to annualise standard deviations we multiply by the **square root** of the number of days in a year. The square root of 256 is exactly 16. Now you can see why I like this number 256 so much.

The strategy risk is very similar but not the same. To compensate for this we need to use **risk adjusted returns** . If a strategy was more risky, an investor could buy the benchmark, add a little bit of leverage, and end up with a risk comparable risk level, and if a strategy was less risky as it is here, the investor could buy a smaller amount of the benchmark to match and have some excess cash to earn interest.

Strictly speaking we should calculate risk adjusted returns using the **Sharpe Ratio** . The Sharpe Ratio accounts for the interest rate we have to pay when using leverage, or any interest we earn from having excess cash. However to keep things simple in this chapter we're going to use a simplified version of the Sharpe Ratio, which is equal to the return divided by the standard deviation.

```
def simple_sharpe_ratio(perc_returns):
    return (perc_returns.mean()*256) /
(perc_returns.std()*16)

simple_sharpe_ratio (strategy_perc_returns )
```

0.657144

```
simple_sharpe_ratio (benchmark_perc_returns)
```

0.489609

Even after accounting for risk the strategy looks better. Another way of demonstrating this is to adjust the benchmark so it has the same risk as the strategy.

```
adjusted_ benchmark_perc_returns =
benchmark_perc_returns * strategy_perc_returns.std() /
benchmark_perc_returns.std()
adjusted_ benchmark_cum_returns =
adjusted_benchmark_perc_returns.cumsum()
```

Remember this ignores the cost of leveraging up the strategy to match the risk of the benchmark, or any interest earned on excess cash because we don't need to put 100% of our money in the benchmark to match the risk of the strategy. This flatters the strategy slightly. Let's recalculate the outperformance, this time by annualising the average difference in returns:

```
diff_cum_returns = strategy_cum_returns -
adjusted_benchmark_cum_returns
diff_returns = diff_cum_returns.diff()
diff_returns .mean()*256
```

0.032538

After adjusting for the lower risk of the strategy, the annual improvement is slightly higher: 3.3% a year. Sounds really good. However can we really believe it? Luck plays a big part in trading, and we should always be asking ourselves whether trading profits are just a fluke.

There is a formal way of testing this known as a 'T-test'. The T-test calculates how likely it is that the mean of the strategy returns is greater than the mean of the benchmark returns. The T-test is

located in another python library, **scipy** :

```
from scipy.stats import ttest_rel
ttest_rel(strategy_perc_returns,
adjusted_benchmark_perc_returns, nan_policy='omit')
```

```
Ttest_relResult(statistic=masked_array(data =
[1.0347721604748643],
             mask = [False],
       fill_value = 1e+20)
, pvalue=masked_array(data =
0.30082053402226544,
             mask = False,
       fill_value = 1e+20)
)
```

The key numbers here are the T statistic (1.03477) and the p-value (0.3082). A high T statistic makes it more likely that the strategy is better than the benchmark. The p-value quantifies how likely. A p-value of 0.3 indicates that there is a 30% chance that the strategy has the same average return as the benchmark. This is a bit better than luck (which would be a 50% chance), but not much more.

Anything under 5% is normally considered a significant result: so this

strategy hasn't passed the test of significance. When using the 5% rule bear in mind the following caveat: if you tested 20 trading strategies out then you'd be likely to find out at least one that was significantly (5%) better than the benchmark. If you have considered and discarded a lot of strategies then you should use a more stringent p-value.

I need to strike another note of caution: the T-test makes some assumptions about the statistical distribution of returns. Assumptions that are normally violently violated when using financial data. To account for this we need to do something different, with a rather grand name: 'Non parametric T-test'.

This test is going to be done using a 'Monte Carlo' technique. It has nothing to do with gambling, although trading and gambling are close cousins, but it has a lot to do with randomness. We're going to randomly generate a large number of 'alternative histories'. Each history will be the same length as the actual backtested data, and consists of points randomly

drawn from the backtest. For each history we measure the difference in returns between the benchmark and the strategy. We then look at the distribution of all the differences. If less than 5% of them are negative (strategy underperforms benchmark) then the strategy will pass the T-test.

```
import numpy as np

monte_carlo_runs = 5000 # make this larger if your computer can cope
length_returns = len(diff_returns.index)
bootstraps = [[int(np.random.uniform(high=length_returns)) for _not_used1 in range(length_returns)] for _not_used2 in range(monte_carlo_runs)]

def average_given_bootstrap(one_bootstrap, diff_returns):
  subset_returns = diff_returns.iloc[one_bootstrap]
  average_for_bootstrap = np.float(subset_returns.mean()*256)
  return average_for_bootstrap

bootstrapped_return_differences = [average_given_bootstrap(one_bootstrap, diff_returns) for one_bootstrap in bootstraps]

bootstrapped_return_differences = pd.Series(bootstrapped_return_differences)
bootstrapped_return_differences.plot.hist(bins=50)
```

Here is a visualisation of the distribution:

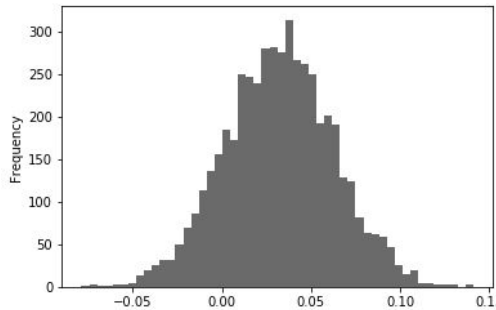


Figure 22-3 Distribution

The average of this distribution is about 3%: in line with the average outperformance of the strategy. But part of the distribution is below 0%, suggesting there is a reasonable chance that the difference in returns could actually be zero or negative. We can check this mathematically:

```
sum(bootstrapped_return_differences < 0)/
float(len(bootstrapped_return_differences))
```

0.1466

Don't worry if you get a slightly different number due to the randomness involved.

We can interpret this as showing that there is a 14.7% probability that the

strategy isn't really outperforming the benchmark. This is a better result than the 30% we got before, using the bog standard T-test, although it still falls short of the 5% critical value that's normally used.

This bootstrapping technique can be used with any kind of statistic: correlation, volatility, Beta… just replace the calculation in the average_given_bootstra p function, and you can get a realistic idea of how accurately we can really measure the relevant statistic.

An alternative method for comparing a strategy and a benchmark is to calculate its **alpha** and **beta** . Apologies for dropping a profusion of Greek letters into the text. These values come from the idea that investors should only care about outperformance after deducting the return that comes from being exposed to the market benchmark. Exposure to the market benchmark is measured by beta. The remaining outperformance is then labelled alpha.

To get technical, the beta is the

covariance of the strategy to the benchmark. To get less technical, a beta of 1.0 indicates you expect to get about the same amount of return as the benchmark. A beta higher than 1.0 implies that you expect to get a higher return, because your strategy is riskier. A beta less than 1.0 suggests you expect a lower return because of a lower exposure to the benchmark, either because your strategy has a lower standard deviation than the benchmark, or because it has a relatively low correlation with the benchmark.

To measure the beta we need to use another fancy statistical technique: **linear regression** . The first step is to check that our benchmark is a sensible one for our strategy. What we're looking for here is a nice linear relationship between the strategy and the benchmark returns. A few outliers aside, that's mostly what we see. There are more sophisticated ways of doing this, but for me nothing beats actually looking at the data.

```
both_returns = pd.concat([strategy_perc_returns,
benchmark_perc_returns], axis=1)
both_weights.columns = both_returns.columns =
```

```
['strategy', 'benchmark']
```

```
both_returns.plot.scatter(x = "benchmark",
y = "strategy")
```

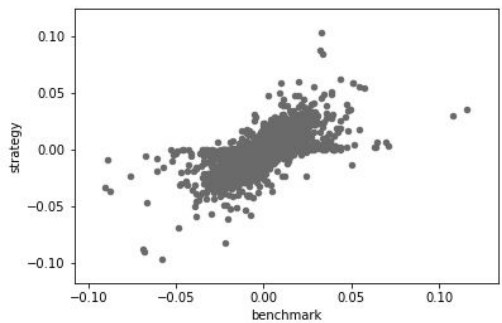


Figure 22-4 Scatter

Because linear regression accounts for different risk levels, we can use the original unadjusted returns here.

```
import statsmodels.formula.api as smf

lm = smf.ols(formula = 'strategy ~ benchmark' ,
data = both_returns).fit()
lm.params
```

```
Intercept    0.000227
benchmark    0.706171
```

The first figure, labelled intercept, is the alpha. The second figure, labelled benchmark, is the value of beta. The alpha is positive, which is good, and works out to around 5.8% a year. But is this just

luck, or statistically significant? Let's check:

```
lm.summary()
```

```
OLS Regression Results
Dep. Variable:
Model:
Method:
Date:
Time:
No. Observations:
Df Residuals:
Df Model:
Covariance Type:

Omnibus:
Prob(Omnibus):
Skew:
Kurtosis:
```

That's a lot of statistics, and I'm not going to try and explain them all to you here. But the key one is the 'P>|t|' column above the 'Intercept' row. That

shows the p-value for a T-test that the intercept value is positive. The value of 0.041, or 4.1%, suggests that there is only a 4.1% chance that the intercept is actually zero or negative. This is under the 5% critical value so we can be reasonably confident that the strategy is better than the benchmark.

Actually these results rely on yet more assumptions: the returns of both the strategy and benchmark need to have nice statistical distributions (which is unlikely), and they need to have a linear relationship (remember we checked this visually, and it looked like it was mostly true). In principle we could use a bootstrap to do this test non-parametrically, but that's out of scope for this chapter.

Let's now return to a question we asked earlier – is the flat relative performance of the strategy after 2005 significant? We can check this using another kind of t-test, which doesn't require the two datasets being tested to match up.

```
from scipy.stats import ttest_ ind
split_date = pd.datetime(2006,1,1)
ttest_ind(diff_returns[diff_returns.index < split_date],
diff_returns[diff_returns.index > = split_date],
nan_policy = 'omit')
```

```
Ttest_indResult(statistic = masked_array(data =
[1.6351376708112633],
             mask = [False],
       fill_value = 1e + 20)
, pvalue = masked_array(data =
0.10207707408174886,
             mask = False,
       fill_value = 1e + 20)
)
```

By now you should be an expert on interpreting these numbers. The p-value (0.102) is reasonably low (though not below 5%), suggesting that there is a good chance that the performance after January 2006 is inferior to the returns before that.

How can we use this information? You might be thinking “Okay, I need to tweak the strategy so it continues to outperform after 2006“. Do not do this! This is an example of what I call **implicit fitting** . Implicit fitting is where you change your trading strategy after looking at all of the results of the backtest.

This is cheating! A backtest is supposed to show us how we could have done in the past. But when we started trading in 1997 we couldn't have possibly known what would happen to the strategy after 2006, unless we had access to a time machine.

Implicit fitting leads to two serious problems. Firstly, we'll probably end up with a trading strategy that's too complicated, will probably be overfitted, and hence will do badly in the future. Secondly, the past performance of our strategy will look better than it really could be.

Since we can't tweak the strategy, let's think about a simple scenario where we're trying to decide how much of our portfolio to allocate to the trading strategy, and how much to put in the benchmark.

First we're going to use implicit fitting. We allocate 100% of our capital to the strategy until January 2006 (since the strategy is much better), and thereafter put 50% in each of the strategy and the

benchmark (since they seem to do equally well after that).

```
strategy_weight =
pd.Series([0.0]*len(strategy_perc_returns), index=
strategy_perc_returns .index)
benchmark_weight =
pd.Series([0.0]*len(benchmark_perc_returns),
index=benchmark_perc_returns.index)

strategy_weight[strategy_weight.index<split_date] =
1.0
benchmark_weight[benchmark_weight.index<split_date]=0.0

strategy_weight[strategy_weight.index>=split_date]
= 0.5
benchmark_weight[benchmark_weight.index>=split_date]=0

both_weights =
pd.concat([strategy_weight,benchmark_weight],
axis=1)
both_returns = pd.concat([strategy_perc_returns,
benchmark_perc_returns], axis=1)
both_weights.columns = both_returns.columns =
['strategy', 'benchmark']

implicit_fit_returns = both_weights*both_returns
implicit_fit_returns = implicit_fit_returns.sum(axis=1)
```

Now we're going to do things properly; making allocation decisions using only past data, without access to a time machine. Allocating capital is quite a complicated business, so to keep things

simple I'm going to use a rule where we allocate in proportion to the historic annual return of each asset, relative to an annual return of 11% (which is roughly the average performance across both the strategy and the benchmark).

Since we know that the strategies relative performance degrades over time, we're going to use the last 5 years of returns to determine that historic mean (to be precise, we're using an exponentially weighted moving average with a 2.5 year half-life, which is the smoother equivalent of a 5 year simple moving average). Using a shorter period would result in weights that are too noisy.

```
rolling_means = pd.ewma(both_returns , halflife =
2.5*256)
rolling_means = rolling_means + (0.16/256)
rolling_means[rolling_means < 0] = 0.000001
total_mean_to_normalise = rolling_means.sum(axis = 1)
total_mean_to_normalise =
pd.concat([total_mean_to_normalise]*2, axis = 1)
total_mean_to_normalise.columns =
rolling_means.columns
rolling_weights = rolling_means /
total_mean_to_normalise
rolling_weights.plot()
```

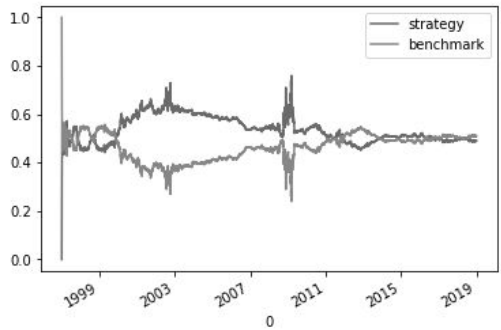


Figure 22-5

These weights don't look that different from the cheating implicit weights; we put most of our capital in the strategy before 2006, and we have something close to an equal split after 2011. However in the period between 2006 and 2011 the rolling mean is still adjusting to the change in relative performance, so we end up putting more in the strategy than the implicit fit does.

```
rolling _fit_returns = rolling_weights*both_returns
rolling _fit_returns = rolling_fit_returns.sum(axis=1)

compare_returns = pd.concat([rolling_fit_returns,
implicit_fit_returns], axis=1)
compare_returns.columns = ['rolling', 'implicit']
compare_returns.cumsum().plot()
```

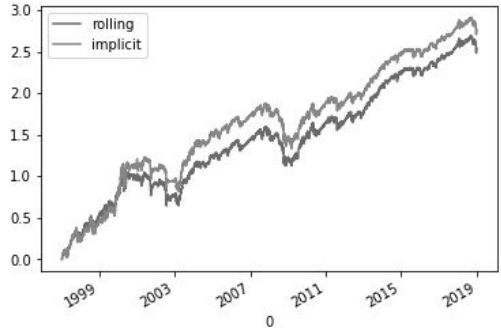


Figure 22-6

```
diff_compare = implicit_fit_returns - rolling_fit_returns
diff_compare.cumsum().plot()
```

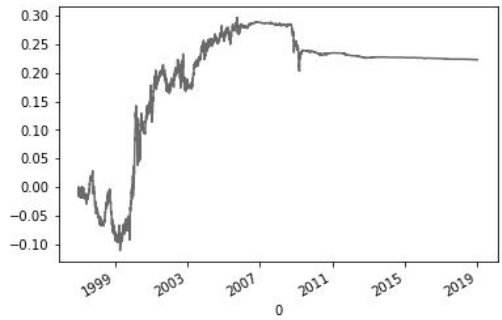


Figure 22-7

Unsurprisingly the implicit fit does better than the rolling fit, with all the outperformance happening before 2008 when it cheats by allocating everything to the strategy. After 2009 both methods are doing pretty much the same thing (in 2008 the implicit fit does slightly better, as it still has a higher weight in the strategy which outperforms the

benchmark).

In hard numbers, if we compare the risk adjusted return:

```
simple_sharpe_ratio(rolling_fit_returns)
```

0.669280152423306

```
simple_sharpe_ratio(implicit_fit_returns)
```

0.6708715830455622

Just by doing a little cheating we've managed to nudge up the Sharpe Ratio. Imagine how much extra we'd get if did some serious fiddling? Before long, you could have a Sharpe Ratio of 2.0, and believe you are a genius trader who can safely use excessive leverage, and afford to pay handsome trading costs to your broker. These are dangerous assumptions which will have a depressing end result: an empty trading account.

Be very careful that you don't head down the path of implicit fitting. Only use data that would really have been available in the past to modify, fit, or allocate to trading strategies.

23 Importing your Data

The most common pitfall when people try to set up their first Python backtesting environment is in getting the data hooked up. This is a non-trivial task, at least if you have to figure it all out by yourself. That's why I will spend a bit of time on the topic and show you some source code that will hopefully help you out and save you some headache.

The first step is of course to acquire the actual data. You need to find a data provider where you can get the data that you need at a price that you are happy with. Rather than giving a suggestion here, I will write about data vendors on my website where it's easier to update and add such information.

If you simply want to get on with it and learn about Python backtesting right away, you can download randomly generated data from my website, at

www.followingthetrend.com/trading-evolved , where I will post data and code related to this book.

In this book, we use the backtesting engine Zipline for all sample code. I had to pick one backtester for the book, and this seemed like a good option. Connecting data to a backtester works differently for every backtesting software. If you would like to use a different software, you would also need to figure out how to get the data hooked up.

Data for stock and for futures work quite differently, and that means that it works a little differently when we import such data. You need a little different code for the two asset classes, and I will start off with importing stock data.

There are two main ways to approach local data management. Which of them you choose depends very much on your own situation, and how you plan to work with your data. You could either just import data directly from whatever format your data provider gives you, into Zipline. Or you could organize all your

data in a local database, and use that as a layer between your data provider and Zipline.

If you are just starting out in the world of backtesting, you probably just want to import the data and get started. Quick and easy.

If on the other hand, you are more advanced, or aspire to be, you may see the benefits of using a local securities database. That way, you can combine data from multiple data providers in a uniform format. You can access the data directly through various tools you can now build with Python, and you can add all kinds of useful metadata information.

Making a Bundle

While making a bundle may seem like something a trader does on a great day in the market, it actually refers to the process of importing your data to Zipline.

This bit is very Zipline specific, and if you decide to use a different backtesting engine, this part won't be applicable.

Other backtesting libraries use their own, different methods for hooking up the data. The method used by Zipline is arguably on the more complex side, but that's the cost of having the added functionality, scalability and feature set of this backtester.

In this chapter, we're going to look at first how to make a bundle for equities and second how to make one for futures. There is quite a lot of similarities, but a few important points where things differ.

For the sample bundles here, we are only getting daily history, but Zipline supports minute level resolution as well and it's quite fast at processing it.

So far in this book, we have been writing code only in **Jupyter Notebook** . That's a great environment for writing, running and analyzing backtests but we need something else to build our bundle. There is another program which was already installed on your computer along with the **Anaconda** package. You should be able to find a program on your computer called **Spyder** . This is a general

purpose Python code editor, where you can write, run and debug code and this is where we will write the bundle.

When making a new bundle, there are two parts to get done. The first part is to actually write the bundle, and that is the bulk of the task and what we're about to go over in detail. The second part, which is easy but not to be forgotten, is that the new bundle has to be registered. That involves adding a tiny bit of text in a text file, but if that's not done Zipline will not know that there's a new bundle in town.

Writing your first bundle can seem a bit daunting, in particular as clear documentation can be difficult to find. As with most things, once you know how, it's not all that hard though.

First a brief overview of what a bundle entails. Clearly a data bundle needs to read data from somewhere. That's probably the simplest part. The **Pandas** library is your friend here, and it can easily read data from local text files, database files, database servers, web calls

or whatever your preference and needs are.

Your bundle needs to have a specific function signature. That just means that there need to be a specific set of arguments to your ingest function. If you're fairly new to all of this, don't worry much about this. You could just use my sample code and change the probably few things that need to be changed to read your own data.

Following the *show, don't tell* guiding principle of this book, let's dig into how you could make a simple but fully functional bundle to read historical equity price data from disk. I will show and explain bit by bit, and then by the end of this section I will show all the code at once.

First the usual import statements.

```
import pandas as pd
from os import listdir
```

We need **Pandas** to read the csv files from disk and for the usual kind of **DataFrame** voodoo, and we need the `listdir` function to check which files are

available.

Next we'll set the path where the data is. Experienced programmers will point out that hardcoding a string like that isn't great coding practice, but this example is not about great coding practice. I want to show you simple ways to get things done, and once you feel more confident in how this works, you can go ahead and improve upon it.

```
# Change the path to where you have your data
path = 'C:\\yourdatapath\\data\\random_stocks\\'
```

Now, here's where the real stuff starts. Next we're going to start off our ingest function. I mentioned briefly that such a function needs to have a specific signature, and that's what you'll see in this function definition.

```
"""
The ingest function needs to have this exact signature,
meaning these arguments passed, as shown below.
"""
def random_stock_data(environ,
                      asset_db_writer,
                      minute_bar_writer,
                      daily_bar_writer,
                      adjustment_writer,
                      calendar,
```

```
        start_session,
        end_session,
        cache,
        show_progress,
        output_dir):
```

This next bit is a demonstration of a neat way of working in Python. Look at the code segment below, and how much we can get done in one short line of code. This line reads all files in the specified folder, strips off the last four characters and returns a list of these trimmed file names.

This of course is based on our assumption here that all the files in the specified folder are csv files, named after a stock ticker, followed by .csv. That is, IBM.csv or AAPL.csv and so on.

```
# Get list of files from path
# Slicing off the last part
# 'example.csv'[:-4] = 'example'
symbols = [f[:-4] for f in listdir(path)]
```

Just in case you forgot to put data in this folder, let's throw an error.

```
if not symbols:
    raise ValueError("No symbols found in folder.")
```

Now we'll prepare three

DataFrame s that we will need to fill up with data in a moment. For now, we're just creating the structure of these **DataFrame** s.

```
# Prepare an empty DataFrame for dividends
divs = pd.DataFrame(columns=['sid',
                             'amount',
                             'ex_date',
                             'record_date',
                             'declared_date',
                             'pay_date']
)
```

```
# Prepare an empty DataFrame for splits
splits = pd.DataFrame(columns=['sid',
                               'ratio',
                               'effective_date']
)
```

```
# Prepare an empty DataFrame for metadata
metadata = pd.DataFrame(columns=('start_date',
                                 'end_date',
                                 'auto_close_date',
                                 'symbol'
                                 'exchange'
                                 )
                        )
```

The Zipline backtester enforced adherence to exchange calendars. While most backtesters simply use the data

provided to them, Zipline will be aware of which days were valid trading days for a given exchange and it will require that these days and no others are populated with historical data.

In reality, the data that vendors supply to you may show some slight deviation from such strict rules. Data might be missing for a certain stock on a valid exchange day, or there could be data provided in error on a non-valid day. Sure, this shouldn't happen, but if you ever find a flawless data provider, please let me know. I sure haven't.

We will define which exchange calendar our bundle should follow when we register it later on, and this bundle code will then be aware of the calendar. If you look again at the function definition signature, you'll see that the calendar was provided there. From that, we can now check which days are valid.

```
    # Check valid trading dates, according to the
selected exchange calendar
    sessions = calendar.sessions_in_range(start_session,
end_session)
```

Now all we have to do is to fetch

the data, align the dates to the calendar, process dividends and metadata. So pretty much the bulk of the actual bundle. But here's where it gets interesting. So far we have been looking at only one function, which we called `random_stock_data`. There are only three more lines for this function.

```
    # Get data for all stocks and write to Zipline
    daily_bar_writer.write(
            process_stocks(symbols, sessions, metadata,
divs)
            )

    # Write the metadata
    asset_db_writer.write(equities=metadata)
```

```
    # Write splits and dividends
    adjustment_writer.write(splits=splits,
                            dividends=divs)
```

If you look at those rows, you should realize that the magic must be happening in a different function, one called `process_stocks`. This is a so called generator function, which will iterate our stocks, process the data and populate the **DataFrames** that we need.

This function will return the historical data needed for the daily bar

writer, and it will populate the metadata and the dividends that we need. We are also providing the empty `spli t` **DataFrame** which we created earlier, as our data is already adjusted for splits as most data tends to be.

That leaves the generator function, which is where data is fetched and processed. After defining the function and passing the list of symbols, the valid trading days, metadata and dividend objects, we initiate a loop of all the stock symbols.

The point of using `enumerat e` to loop the stocks is that we will automatically get a number for each stock, in order. The first one will be 0, then 1 and so on. We need a unique security ID (SID) for each stock, and as long as it's a number and unique all is fine. These automatically increasing numbers that enumerate gives us will do perfectly.

```
"""
Generator function to iterate stocks,
build historical data, metadata
and dividend data
"""
```

```
def process_stocks(symbols, sessions, metadata, divs):
    # Loop the stocks, setting a unique Security ID (SID)
    for sid, symbol in enumerate(symbols):
```

The remainder of the function is inside that loop, meaning that all the rest of the code will be run once for every stock symbol available in our data folder. Depending on much data you are supplying, this may take a few minutes. Whenever a task might take a while, it can be useful to output some sort of progress so that we know all is proceeding as it should. While this can be done much prettier, I'll use a simple print statement here.

```
        print('Loading {}...'.format(symbol))
        # Read the stock data from csv file.
        df = pd.read_csv('{}/{}.csv'.format(path,
symbol), index_col = [0], parse_dates = [0])
```

As you see in that code, we read the data from file using the **Pandas** library, specifying the path that we set earlier, and adding the symbol to create a file name. We set the first column as the index, and ask **Pandas** to parse the date format.

Next we will make sure that our

data conforms to the specified exchange calendar. As a side note, once you get more comfortable with Python and Zipline, you can try editing calendars or creating your own.

When aligning the dates to the calendar, you need to decide what to do in case of a discrepancy. Hopefully there won't be any difference, but there need to be some sort of handling if that occurs. In the code below, as an example, I told Pandas to forward fill missing days, and then to drop any potential none values as a failsafe.

Note that the code below checks the first and last date of the actual data we read from disk, and then reindexes this data to use the valid exchange days in that range.

```
    # Check first and last date.
    start_date = df.index[0]
    end_date = df.index[-1]
```

```
    # Synch to the official exchange calendar
    df = df.reindex(sessions.tz_localize(None))
[start_date:end_date]
```

```
    # Forward fill missing data
    df.fillna(method='ffill', inplace=True)
```

```
    # Drop remaining NaN
    df.dropna(inplace=True)
```

Now we have enough information on the stock for the metadata. You could actually add more information to the metadata if you like, such as company name for instance, but we have everything required for things to work. We just add to the metadat a **DataFrame** which we passed in the function definition.

A common pitfall is to overlook the field for exchange. After all, we don't really need that information and won't use it for anything. The problem however is that if you omit that field, Zipline will not be able to use your bundle. Here we simply hardcode it, to avoid the issue.

```
    # The auto_close date is the day after the last
trade.
    ac_date = end_date + pd.Timedelta(days=1)
```

```
    # Add a row to the metadata DataFrame.
    metadata.loc[sid] = start_date, end_date, ac_date,
symbol, "NYSE"
```

The last task before we pass the data back is to check for dividends. As you may or may not have dividend data available, I added a check for if such a column is in the data file. If so, we process it and add the data to the div s **DataFrame** .

```
    # If there's dividend data, add that to the
dividend DataFrame
    if 'dividend' in df.columns:
```

```
        # Slice off the days with dividends
        tmp = df[df['dividend'] != 0.0]['dividend']
        div = pd.DataFrame(data=tmp.index.tolist(),
columns=['ex_date'])
```

```
        # Provide empty columns as we don't have this
data for now
        div['record_date'] = pd.NaT
        div['declared_date'] = pd.NaT
        div['pay_date'] = pd.NaT
```

```
        # Store the dividends and set the Security ID
        div['amount'] = tmp.tolist()
        div['sid'] = sid
```

```
        # Start numbering at where we left off last
time
        ind = pd.Index(range(divs.shape[0],
divs.shape[0] + div.shape[0]))
```

```
        div.set_index(ind, inplace=True)
```

```
        # Append this stock's dividends to the list of
all dividends
        divs = divs.append(div)
```

And that leaves only the final little detail of passing the historical data back to the caller, and since this is a generator function, we use yiel d .

```
    yield sid, df
```

That is the entire bundle. While I do admit that figuring out how to do this from simply reading the online documentation can seem a little daunting if you are not already a Python programmer, using my sample here and adapting it to your own data should be a breeze.

Here is again the entire code for this data bundle.

```
import pandas as pd
from os import listdir

# Change the path to where you have your data
path = 'C:\\Users\\Andreas Clenow\\BookSamples\
\BookModels\\data\\random_stocks\\'
```

```
"""
The ingest function needs to have this exact signature,
meaning these arguments passed, as shown below.
"""
def random_stock_data(environ,
                      asset_db_writer,
                      minute_bar_writer,
                      daily_bar_writer,
                      adjustment_writer,
                      calendar,
                      start_session,
                      end_session,
                      cache,
                      show_progress,
                      output_dir):
```

```
    # Get list of files from path
    # Slicing off the last part
    # 'example.csv'[:-4] = 'example'
    symbols = [f[:-4] for f in listdir(path)]
```

```
    if not symbols:
        raise ValueError("No symbols found in folder.")
```

```

```

```
    # Prepare an empty DataFrame for dividends
    divs = pd.DataFrame(columns=['sid',
                                 'amount',
                                 'ex_date',
                                 'record_date',
                                 'declared_date',
                                 'pay_date']
```

```
    )
```

```
    # Prepare an empty DataFrame for splits
    splits = pd.DataFrame(columns=['sid',
                                   'ratio',
                                   'effective_date']
    )
```

```
    # Prepare an empty DataFrame for metadata
    metadata = pd.DataFrame(columns=('start_date',
                                     'end_date',
                                     'auto_close_date',
                                     'symbol',
                                     'exchange'
                                     )
                                )


    # Check valid trading dates, according to the
selected exchange calendar
    sessions = calendar.sessions_in_range(start_session,
end_session)
```

```
    # Get data for all stocks and write to Zipline
    daily_bar_writer.write(
            process_stocks(symbols, sessions, metadata,
divs)
            )

    # Write the metadata
    asset_db_writer.write(equities=metadata)
```

```
```

```
    # Write splits and dividends
    adjustment_writer.write(splits=splits,
                            dividends=divs)
```

```
"""
Generator function to iterate stocks,
build historical data, metadata
and dividend data
"""
def process_stocks(symbols, sessions, metadata, divs):
    # Loop the stocks, setting a unique Security ID (SID)
    for sid, symbol in enumerate(symbols):
```

```
        print('Loading {}...'.format(symbol))
        # Read the stock data from csv file.
        df = pd.read_csv('{}/{}.csv'.format(path,
symbol), index_col=[0], parse_dates=[0])
```

```
        # Check first and last date.
        start_date = df.index[0]
        end_date = df.index[-1]
```

```
        # Synch to the official exchange calendar
        df = df.reindex(sessions.tz_localize(None))
[start_date:end_date]
```

```
        # Forward fill missing data
        df.fillna(method='ffill', inplace=True)
```

```
        # Drop remaining NaN
```

```
        df.dropna(inplace=True)

        # The auto_close date is the day after the last
trade.
        ac_date = end_date + pd.Timedelta(days=1)

        # Add a row to the metadata DataFrame. Don't
forget to add an exchange field.
        metadata.loc[sid] = start_date, end_date, ac_date,
symbol, "NYSE"

        # If there's dividend data, add that to the
dividend DataFrame
        if 'dividend' in df.columns:

            # Slice off the days with dividends
            tmp = df[df['dividend'] != 0.0]['dividend']
            div = pd.DataFrame(data=tmp.index.tolist(),
columns=['ex_date'])

            # Provide empty columns as we don't have this
data for now
            div['record_date'] = pd.NaT
            div['declared_date'] = pd.NaT
            div['pay_date'] = pd.NaT

            # Store the dividends and set the Security ID
            div['amount'] = tmp.tolist()
            div['sid'] = sid

            # Start numbering at where we left off last
```

```
time
        ind = pd.Index(range(divs.shape[0],
divs.shape[0] + div.shape[0]))
        div.set_index(ind, inplace=True)

        # Append this stock's dividends to the list of
all dividends
        divs = divs.append(div)

    yield sid, df
```

This code should be saved as a .p y file in your Zipline bundle folder. Exactly where yours is located on your computer depends on your own installation and local setup. You should easily find it by searching, but for reference, mine is located at the path shown just below. For the example here, I will assume that you saved this bundle as random_stock_data.p y . You can call the file whatever you like, but we need to refer to it in a moment when we register the bundle.

```
C:\ProgramData\Anaconda3\envs\zip35\Lib\site-
packages\zipline\data\bundles
```

Now that we have constructed the bundle, we need to register it with Zipline. That's done in a file called extention.p y which you should find in your

home directory, under /.ziplin e . If you are on Windows, that would probably be under c:/users/username/.zipline/extension.p y and if you are one a UNIX style OS, it would be ~/.zipline/extension.p y . If the file is not there, create it.

In this file, we need to import the bundle, and register it, as per the syntax below, assuming that you saved the bundle code in the correct folder, using the name random_stock_data.p y .

```
from zipline.data.bundles import register,
random_stock_data
register(' random_stock_data ', random_stock_data .
random_stock_data ,
        calendar_name = 'NYSE')
```

As you see here, this is where we specify the exchange calendar, which as we saw earlier, defines which days are valid trading days. Perhaps you are wondering why we are repeating the name twice, random_stock_data.random_stock_dat a . Well, that simply because in this example, the name of the file and the name of the function in the file happen to be the same. We're also naming the bundle using the same name here, to keep

it simple.

After this file is modified and saved, we're ready to ingest the bundle. The ingest process refers to the process of actually running the bundle code to import the data into Zipline.

Now you are ready to ingest your new bundle. Remember how we did that earlier, in chapter 7?

Open a terminal window for your zip3 5 environment. You can do that through **Anaconda Navigator** . Then ingest the new bundle just like before.

```
zipline ingest – b random_equities
```

If all now worked as it should, you will see how the data is pulled in, stock by stock, and stored by Zipline. When this process is complete, you are ready to start constructing some serious, albeit somewhat random backtests.

As you have seen in earlier chapters, each backtest code for each model specifies which bundle it pulls data from. So now you can simply modify that part of the code, usually near the bottom,

to use your random bundle.

Zipline and Futures Data

The landscape of Python backtesting engines is fast moving, and perhaps by the time you read this something revolutionary has occurred, rending part of this book obsolete. But as of writing this, Zipline is in my view leading the race in terms of providing the most robust and feature rich Python backtesting environment for futures. We already discussed this a bit in chapter 14.

It does however require a little work to set up properly, in a local installation. But fear not, I will guide you through it. Since you made it all the way to this chapter, you have probably already learned how to make a Zipline bundle for equities, as we discussed earlier in this chapter. Much of what we learned there is valid for futures as well, but there are a few tricky points to navigate.

The thing you always have to be aware of in Python world, is that unfortunately, nothing is really finished, polished software. You will often encounter situations where you wonder why someone left something seemingly unfinished, or why you would need to go edit someone else's source code just to get your solution working. That's just Python world for you. But on the other hand, it's all free.

As an overview to start with, here is what we need to get done in order to be up and running with futures backtesting for Zipline:

- Construct a futures bundle.
- Provide accurate futures metadata to bundle.
- Provide additional meta data for the futures root symbols.
- Register the futures bundle in extension.p y .
- Edit constants.p y to make sure all your markets have default definitions for slippage and exchange fees.

A few pitfalls to be careful with:

- Zipline expects a very specific syntax for futures symbol. Make sure you format the

symbol as **RRMYY** , where **RR** is a two character root symbol, **M** is a one character month code, and **YY** is a two digit year. Example: **CLF02** for January 2002 crude contract.

- Note that a two character root symbol is required. Not one, three or four, as may be the case in reality. So you need to enforce the two character syntax.
- All market root symbols have to be listed in the file constants.p y with a default fee and slippage. If you want to include a market that's not already in that file, you need to add it.
- As for stock bundles, Zipline expects to be provided with a **DataFrame** for dividend and split adjustments. Clearly that does not make much sense, but as long as we simply provide an empty **DataFrame** , the backtester is happy.
- Zipline expects data to be provided on the exact days specified by their holiday calendar, and the data ingest will fail if your data lacks days or have excess days. It's possible and even likely that this is the case, so you need to make use of Pandas to ensure that your dates match the expected dates. This is the same as

for the stocks bundle we saw earlier, and the solution is the same.

- The current version of Zipline, which is 1.3 as of writing this, has some issues with data earlier than the year 2000. Restricting data to post 2000 simplifies the process.
- If you have data available for first notice day, you may want to incorporate this in the bundle. As many readers likely don't, I will simplify and skip it for now. You could approximate the first notice day for most commodities by setting auto close day one month before last traded date.

Futures Data Bundle

In large part, a futures bundle works the same as an equities bundle, but pay attention to the special considerations for this asset class. In the sample bundle here, we will read the random data which you can download from the book website.

I provide this random data to help you get up and running faster. By having this random sample data set and the bundles for them, you should be able to

see the logic, test it and modify where needed to get your own data working.

To run this sample, you would either need to download my random sample data from the book website, **www.followingthetrend.com/trading-evolved/** , or modify the code to use your own data. Apart from the random data, there is also a metadata lookup file that you can download from the site, which holds important futures information such as point value, sector and more. This bundle uses that file as well, providing the information to the Zipline framework.

I'm also going to introduce you to a new little trick in this code sample. Loading the data might take a while, and spitting out a new text row for every market like we did in the equities bundle is a little primitive. This time, you'll get a neat little progress bar instead.

Starting off on the code, first we have the usual import statements, including the **tqdm** library that we'll use for the progress bar.

```
import pandas as pd
```

```
from os import listdir
from tqdm import tqdm # Used for progress bar
```

As mentioned, both random historical data and metadata lookup is provided. The next piece of the code specifies where that data can be located and then reads the metadata lookup to memory, so that we can access is later. Change the path to match the location where you put the downloaded data.

```
# Change the path to where you have your data
base_path = "C:/your_path/data/"
data_path = base_path + 'random_futures/'
meta_path = 'futures_meta/meta.csv'
futures_lookup = pd.read_csv(base_path + meta_path,
index_col=0)
```

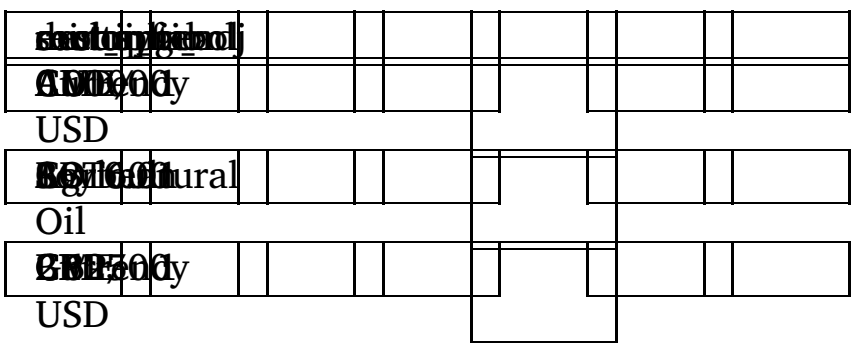
The layout of my futures lookup table is shown in Table 23.1. Most of these fields are self-explanatory, such as the root symbol, description and sector. But you may be wondering about the multiplier and the minor fx adjustment.

The multiplier, sometimes referred to as point value or contract size, is a key property of a futures contract. It defines how many dollars you gain or lose if the contract price changes by one dollar.

Thereby the name multiplier.

The minor fx adjustment factor is something that I included as a reminder. Some US futures markets are quoted in US cents, rather than US Dollars, and if you don't adjust for that you may get some odd results. You should check with your local data provider to see if they supply USD or USc pricing for such markets.

Table 23.1 Futures Lookup Table



We start off the ingest function with the same signature as you've seen in the equity sample bundle.

```
"""
The ingest function needs to have this exact signature,
meaning these arguments passed, as shown below.
"""
def random_futures_data(environ,
                        asset_db_writer,
                        minute_bar_writer,
```

```
            daily_bar_writer,
            adjustment_writer,
            calendar,
            start_session,
            end_session,
            cache,
            show_progress,
            output_dir):
```

So far, it's not really different from what we saw earlier. The next segment here is also very similar, but note that we have to provide a few additional fields in the metadata.

```
# Prepare an empty DataFrame for dividends
divs = pd.DataFrame(columns=['sid',
                             'amount',
                             'ex_date',
                             'record_date',
                             'declared_date',
                             'pay_date']
)
```

```
# Prepare an empty DataFrame for splits
splits = pd.DataFrame(columns=['sid',
                               'ratio',
                               'effective_date']
)

# Prepare an empty DataFrame for metadata
metadata = pd.DataFrame(columns=('start_date',
                                 'end_date',
                                 'auto_close_date',
```

```
                    'symbol',
                    'root_symbol',
                    'expiration_date',
                    'notice_date',
                    'tick_size',
                    'exchange'
                    )
            )
```

For futures contracts, we need to supply information about root symbol, expiration date, notice date and tick size, and we'll do that in the metadata.

The next segment hold no new surprises, and this is where we get the valid trading days for the selected calendar, call the function that will fetch and process the data, and write the result. Of course, it's in that particular function where the interesting part happens and we'll soon look closer at it.

Note that we also have to write the empty data for splits and dividends. We clearly have no such information on futures markets, so just provide empty frames with the expected headers.

```
    # Check valid trading dates, according to the
selected exchange calendar
    sessions = calendar.sessions_in_range(start_session,
```

```
end_session)

    # Get data for all stocks and write to Zipline
    daily_bar_writer.write(
            process_futures(symbols, sessions, metadata)
            )
```

```
    adjustment_writer.write(splits = splits,
dividends = divs)
```

You may remember earlier that the last step was to write the metadata. This works almost the same here, and as before the generator function, here called process_future s , has prepared the contract metadata for us.

We now need to prepare metadata on the root level, and write this to the Zipline framework. We can use the futures lookup table that we got previously, almost as is. We just need to add a column with a unique root_symbol_i d as well as delete the now unneeded minor_fx_ad j field.

```
    # Prepare root level metadata
    root_symbols = futures_lookup.copy()
    root_symbols['root_symbol_id'] =
root_symbols.index.values
    del root_symbols['minor_fx_adj']
```

```
    #write the meta data
```

```
    asset_db_writer.write(futures=metadata,
root_symbols=root_symbols)
```

That's all of the ingest function, but we still haven't looked at the function which actually reads and processes the data. The structure of this is the same as for the equities bundle, but pay attention to the special considerations with futures.

First we define the function, and start a loop of all symbols. Note how we use **tqdm** here when we initiate the loop. That's all we need to do in order to get a nice little progress bar to display during this loop.

```
def process_futures(symbols, sessions, metadata):
    # Loop the stocks, setting a unique Security ID (SID)
    sid = 0
```

```
    # Loop the symbols with progress bar, using tqdm
    for symbol in tqdm(symbols, desc='Loading
data...'):
        sid += 1
```

```
        # Read the stock data from csv file.
        df = pd.read_csv('{}/{}.csv'.format(data_path,
symbol), index_col=[0], parse_dates=[0])
```

Now that we have read the data from disk, we can start doing some

processing on it. First, I'll check the minor fx adjustment factor and multiply all prices with that.

```
    # Check for minor currency quotes
    adjustment_factor = futures_lookup.loc[
        futures_lookup['root_symbol'] == df.iloc[0]
['root_symbol']
        ]['minor_fx_adj'].iloc[0]
```

```
    df['open'] *= adjustment_factor
    df['high'] *= adjustment_factor
    df['low'] *= adjustment_factor
    df['close'] *= adjustment_factor
```

The syntax used above `value *= x`, is the same as `value = value * x`, just like `value += x` is the same as `value = value + x`.

Getting perfect data can be a near futile enterprise. I find that one very common issue with futures data is that once in a while, you get a high value lower than close, or some similar issue. Just to show you a method to prevent a small error like that to blow up your code, I included this segment below.

```
    # Avoid potential high / low data errors in data
set
    # And apply minor currency adjustment for USc
quotes
```

```
    df['high'] = df[['high', 'close']].max(axis=1)
    df['low'] = df[['low', 'close']].min(axis=1)
    df['high'] = df[['high', 'open']].max(axis=1)
    df['low'] = df[['low', 'open']].min(axis=1)
```

Now we re-index the dates to match the valid session dates, and cut dates before 2000 to quickly bypass a current issue with Zipline with such dates.

```
    # Synch to the official exchange calendar
    df = df.reindex(sessions.tz_localize(None))
[df.index[0]:df.index[-1] ]
```

```
    # Forward fill missing data
    df.fillna(method='ffill', inplace=True)
```

```
    # Drop remaining NaN
    df.dropna(inplace=True)

    # Cut dates before 2000, avoiding Zipline issue
    df = df['2000-01-01':]
```

We need to gather some metadata for each contract, and to make the code easier to read and manage, I've outsourced that to a separate function, which we call here. We'll get to the details of that function in a moment.

```
    # Prepare contract metadata
    make_meta(sid, metadata, df, sessions)
```

Finally, we finish off this symbol loop as well as the function by deleting the fields we no longer need and returning the security id and the data.

```
        del df['openinterest']
        del df['expiration_date']
        del df['root_symbol']
        del df['symbol']

        yield sid, df
```

That just leaves the metadata construction, which we left in a separate function. This function adds a row to our metadat a **DataFrame** for each individual contract.

```
def make_meta(sid, metadata, df, sessions):
        # Check first and last date.
        start_date = df.index[0]
        end_date = df.index[-1]

        # The auto_close date is the day after the last
trade.
        ac_date = end_date + pd.Timedelta(days=1)

        symbol = df.iloc[0]['symbol']
        root_sym = df.iloc[0]['root_symbol']
        exchng =
futures_lookup.loc[futures_lookup['root_symbol'] ==
```

```
root_sym ]['exchange'].iloc[0]
        exp_date = end_date

        # Add notice day if you have.
        # Tip to improve: Set notice date to one month
prior to
        # expiry for commodity markets.
        notice_date = ac_date
        tick_size = 0.0001   # Placeholder

        # Add a row to the metadata DataFrame.
        metadata.loc[sid] = start_date, end_date, ac_date,
symbol, \
                            root_sym, exp_date, notice_date,
tick_size, exchng
```

This function, make_met a , populates values for each contract, adding root symbol, start and end dates etc. This is a pretty straight forward function, but there's one interesting point to mention here.

Keeping things simple, in this code I simply set the first notice date to the same as the expiry date. If you only trade financial futures, that's not really a problem, but with commodity markets it might be.

If your data provider has the actual first notices dates, just provide those. But

for most readers, that's probably not the case. So I will make a suggestion, and give you a little bit of a homework. If you made it all the way here in the book, you should be able to figure this one out.

What you can do to approximate the first notice date, is to set it to one month prior to expiry, if it is a commodity market. So what you'd need to do is to check the sector in the lookup table, and if it's commodity, deduct a month from the expiry date. You'll figure it out.

For convenience, here's the full source code for this futures bundle.

```
import pandas as pd
from os import listdir
from tqdm import tqdm # Used for progress bar

# Change the path to where you have your data
base_path = "C:/your_path/data/"
data_path = base_path + 'random_futures/'
meta_path = 'futures_meta/meta.csv'
futures_lookup = pd.read_csv(base_path + meta_path,
index_col=0)

"""
The ingest function needs to have this exact signature,
meaning these arguments passed, as shown below.
"""
```

```
def random_futures_data(environ,
                        asset_db_writer,
                        minute_bar_writer,
                        daily_bar_writer,
                        adjustment_writer,
                        calendar,
                        start_session,
                        end_session,
                        cache,
                        show_progress,
                        output_dir):
```

```
    # Get list of files from path
    # Slicing off the last part
    # 'example.csv'[:-4] = 'example'
    symbols = [f[:-4] for f in listdir(data_path)]
```

```
    if not symbols:
        raise ValueError("No symbols found in folder.")
```

```
    # Prepare an empty DataFrame for dividends
    divs = pd.DataFrame(columns=['sid',
                                 'amount',
                                 'ex_date',
                                 'record_date',
                                 'declared_date',
                                 'pay_date']
    )
```

```
    # Prepare an empty DataFrame for splits
    splits = pd.DataFrame(columns=['sid',
                                   'ratio',
```

```
                        'effective_date']
    )

    # Prepare an empty DataFrame for metadata
    metadata = pd.DataFrame(columns=('start_date',
                                     'end_date',
                                     'auto_close_date',
                                     'symbol',
                                     'root_symbol',
                                     'expiration_date',
                                     'notice_date',
                                     'tick_size',
                                     'exchange'
                                     )
                            )

    # Check valid trading dates, according to the
selected exchange calendar
    sessions = calendar.sessions_in_range(start_session,
end_session)

    # Get data for all stocks and write to Zipline
    daily_bar_writer.write(
            process_futures(symbols, sessions, metadata)
            )
```

```
    adjustment_writer.write(splits=splits,
dividends=divs)
```

```
    # Prepare root level metadata
    root_symbols = futures_lookup.copy()
    root_symbols['root_symbol_id'] =
root_symbols.index.values
```

```
    del root_symbols['minor_fx_adj']
```

```
    #write the meta data
    asset_db_writer.write(futures=metadata,
root_symbols=root_symbols)
```

```
def process_futures(symbols, sessions, metadata):
    # Loop the stocks, setting a unique Security ID (SID)
    sid = 0
```

```
    # Loop the symbols with progress bar, using tqdm
    for symbol in tqdm(symbols, desc='Loading
data...'):
        sid += 1
```

```
        # Read the stock data from csv file.
        df = pd.read_csv('{}/{}.csv'.format(data_path,
symbol), index_col=[0], parse_dates=[0])
```

```
        # Check for minor currency quotes
        adjustment_factor = futures_lookup.loc[
            futures_lookup['root_symbol'] == df.iloc[0]
['root_symbol']
            ]['minor_fx_adj'].iloc[0]
```

```
        df['open'] *= adjustment_factor
        df['high'] *= adjustment_factor
        df['low'] *= adjustment_factor
        df['close'] *= adjustment_factor

        # Avoid potential high / low data errors in data
```

```
set
        # And apply minor currency adjustment for USc
quotes
        df['high'] = df[['high', 'close']].max(axis=1)
        df['low'] = df[['low', 'close']].min(axis=1)
        df['high'] = df[['high', 'open']].max(axis=1)
        df['low'] = df[['low', 'open']].min(axis=1)

        # Synch to the official exchange calendar
        df = df.reindex(sessions.tz_localize(None))
[df.index[0]:df.index[-1] ]
```

```
        # Forward fill missing data
        df.fillna(method='ffill', inplace=True)
```

```
        # Drop remaining NaN
        df.dropna(inplace=True)
```

```
        # Cut dates before 2000, avoiding Zipline issue
        df = df['2000-01-01':]
```

```
        # Prepare contract metadata
        make_meta(sid, metadata, df, sessions)
```

```
        del df['openinterest']
        del df['expiration_date']
        del df['root_symbol']
        del df['symbol']
```

```
        yield sid, df
```

```
```

```
def make_meta(sid, metadata, df, sessions):
        # Check first and last date.
        start_date = df.index[0]
        end_date = df.index[-1]

        # The auto_close date is the day after the last
trade.
        ac_date = end_date + pd.Timedelta(days=1)
```

```
        symbol = df.iloc[0]['symbol']
        root_sym = df.iloc[0]['root_symbol']
        exchng =
futures_lookup.loc[futures_lookup['root_symbol'] ==
root_sym ]['exchange'].iloc[0]
        exp_date = end_date
```

```
        # Add notice day if you have.
        # Tip to improve: Set notice date to one month
prior to
        # expiry for commodity markets.
        notice_date = ac_date
        tick_size = 0.0001   # Placeholder

        # Add a row to the metadata DataFrame.
        metadata.loc[sid] = start_date, end_date, ac_date,
symbol, \
                            root_sym, exp_date, notice_date,
tick_size, exchng
```

We also need to register this bundle, which just as we saw earlier with the equity bundle, we do in the file extension.p y . Assuming that you already

created or modified this file earlier for the random equity bundle, your new file should look something like this, given that you just saved the futures bundle as random_futures.p y in the Zipline bundles folder.

```
from zipline.data.bundles import register,
random_equities, random_futures
register('random_equities',
random_equities.random_bundle,
        calendar_name = 'NYSE')
register('random_futures',
random_futures.futures_bundle,
        calendar_name = 'us_futures')
```

This will register the bundle, so that we can ingest it. That is, now we can pull the data from the comma separated files on disk and make them available for Zipline to use.

If you are reading this in the paperback, you have some typing to do here. Or you can just go to **www.followingthetrend.com/trading-evolved** and download it. If you are reading this on a Kindle, you should be able to copy paste the text, and if you are reading this on a pirate copy PDF I will make use of my very particular set of

skills, acquired over a very long career, and I will find you.

This file, as the source code for the bundle shows, can be easily read by **Pandas** and used to provide Zipline with the required information about each market.

Finally, unless this issue has been fixed by the time you read this, locate the file constants.p y . It will be faster for you to search for the file than to type a path from this book. In this file, you have two dictionaries that need modification. The first, called FUTURES_EXCHANGE_FEES_BY_SYMBO L lists each futures root and the corresponding exchange fee. Make sure all markets you intend to cover are listed there. If not, add them.

The second dictionary in the same file is called ROOT_SYMBOL_TO_ET A and it impacts slippage conditions. Again, make sure that all markets you intend to cover in your backtesting are listed here.

Now you are ready to ingest the bundle.

```
zipline ingest -b random_futures
```

Patching the Framework

As of writing this, there is a missing line of code in the Zipline framework which will prevent your futures data from loading. Hopefully this is already fixed by the time you read this book, but otherwise you will have to make the necessary modification yourself. Don't worry, it's not difficult, once you know what to do.

What you need to do is to locate a file called `run_algo.p y` , which can be found under your Zipline installation, in the subfolder `util s` , and make a minor modification. Search for this file if you can't find it right away, as the path may be slightly different on your computer than on mine.

Open this file in **Spyder** , and scroll down to around line number 160, where you will find this code.

```
    data = DataPortal(
```

```
        env.asset_finder,
        trading_calendar=trading_calendar,
        first_trading_day=first_trading_day,

equity_minute_reader=bundle_data.equity_minute_bar_reader,

equity_daily_reader=bundle_data.equity_daily_bar_reader,

adjustment_reader=bundle_data.adjustment_reader,
    )
```

We need to one line to this code, so that it looks like this code below. After that, Zipline and your futures data should get along famously.

```
    data = DataPortal(
        env.asset_finder,
        trading_calendar=trading_calendar,
        first_trading_day=first_trading_day,

equity_minute_reader=bundle_data.equity_minute_bar_reader,

equity_daily_reader=bundle_data.equity_daily_bar_reader,

future_daily_reader=bundle_data.equity_daily_bar_reader,

adjustment_reader=bundle_data.adjustment_reader,
    )
```

Well, I did warn you early on that nothing is really finished in Python world. You may be required to modify someone else's code at times to get things to work

as you'd expect them to.

24 Data and Databases

Data is probably the biggest single issue when it comes to financial modeling and backtesting. No matter how good your trading algorithm is, if it's based on flawed data, it's a waste of time. There are two primary aspects of data. There is the quality of the data, and there is the coverage of data. Quality refers to how reliable it is. Generally speaking, free data will be of lower quality than expensive data.

Quality issues can be of different kinds and severity. There could be mis-ticks introduced, making it seems like there was a massive spike in the price, where none happened in reality, or perhaps a zero value was slipped in to

make it look like a stock suddenly went bankrupt. Data could be missing, or a **NaN** value being fed to you all of a sudden. Perhaps adjustments for splits are sometimes missing or incorrect, resulting in your algorithm going haywire. There could, potentially, be any sort of issue.

While these kind of quality issues are more common in freely available data sources, they are by no means unheard of in expensive, curated time-series databases. The pursuit of quality data is an ongoing struggle.

The other aspect is somewhat easier to deal with, since it usually just requires money. The other aspect is coverage, both in terms of instruments covered and in terms of type of data available.

It's primarily on that second aspect where you will soon realize that freely available data sources don't cut the mustard.

Equity data makes for the best example of why this is the case. It may well appear as if free online data sources have pretty good coverage of equity data.

Even after Yahoo and Google decided to shut down their API access, there are still quite a few places where you can get at least daily history for stocks.

But if you look a little closer and consider what you really need, you will find that these free sources don't have what it takes. Free or low cost data is usually adjusted for splits and corporate actions. After all, without adjusting for this, the data would be utterly useless.

Most of these free or low cost databases lack dividend information, and while that may not seem like a big deal, it can have a large impact. In the long run, the effects of dividends can be quite substantial. But even in the shorter run, you can have some real distortions to your models. If your backtest holds a stock while it goes ex-div, and you lack the data or logic to handle that, your backtester will think that you just lost money. Dividend information, and the logic to handle it, is absolutely necessary to make a proper equity simulation.

But as important as dividends are,

there is another much more important point. To build realistic backtests, you need to have data available, as it looked back in time. One thing that these free and low cost sources tend to lack is stocks that are delisted, merged or otherwise changed. The stocks that are currently available are the ones that did well enough to survive. Thereby the term survivorship bias.

Remember how many tech stocks that went the way of the dodo by the end of the 90s. Those stocks that went belly up back then will now be gone. Purged from reality, as far as the low cost data sources are concerned. They never existed.

Clearly, that will create a bias in your backtests. The only stocks available to trade are the ones that performed well enough to exist today. Ignoring this issue will almost certainly give you a false sense of security in that the backtest will show far better results than reality would. Therefore, all delisted stocks, the so called graveyard, need to be included.

A closely related issue to this is

some sort of information to decide which stocks you would have been likely to consider in the past. As of writing this, Apple is the largest company in the world. Everyone knows that this is a very large company which had a stellar performance in the past. But if your backtest goes back twenty or thirty years, the situation was quite different.

I'm old enough to remember when Apple nearly went bankrupt and had to ask Bill Gates for help with paying the bills. Back then, when Apple looked like a tiny little computer manufacturer of little consequence, it would probably not have been on your radar. Making the assumption that your backtest would trade it decades ago, just because it's a big stock now, is a mistake.

There are two common ways of constructing a dynamic investment universe, to alleviate this issue. The first would be to use historical data on market capitalization and/or liquidity. By using this, you can identify which stocks were large enough or liquid enough to have been likely candidates at a point far back

in time, and your code can then make a dynamic list of stocks to trade.

The other method would be to pick an index, and look at the historical members of that index. For that, you don't need additional liquidity or market capitalization data for the stocks, but just information on entries and exits of an index.

The index method is likely to be easier for readers of this book, and you will find more or less equivalent end result, and I used that for equity portfolio model in this book.

What I want to say with all of this is that you will very quickly reach the end of the line for what can be done with free data. That leaves you with the task of figuring out what data source you want to use, and to connect it to your backtester.

In the previous chapter, we looked at ways to read custom data from local text files. That's a good entry into the world of custom data, but as you get more advanced you're likely to acquire more sophisticated needs.

Once you get into more advanced aspects of data, and perhaps start combining data from various different sources, it makes sense to set up your own local securities database. This of course is by no means a requirement, and if you are still struggling with some technophobia, you can safely skip this chapter.

Your Very Own Securities Database

What I would recommend is to store your data in a proper database, regardless of what format your chosen data provider gives to you.

Now we are slightly beyond the core topic of trading, and sliding into a little more technical subject. But let me explain why.

Assume that your data provider, for example, delivers CSV flat files to you daily with ten fields per stock in a particular layout. You could now make your backtester read this data, by

explaining exactly where this file is located, what the naming schemes are and how the fields are laid out. You probably have other software tools on your computer that you want to use to access this data, and you could do the same there, have them read the files directly. Your backtester is probably not the only piece of software that you will be using to access this data.

The problem comes the day that your data provider changes something, or when you decide to change provider. And even if none of those happen, you may run into issues when you want to add a second data provider or add information such as code lookup, sector info and other metadata.

What you probably want to do instead, is to put your own securities database in between the data providers and your tools. And yes, dear reader, that means that you have to learn some basics about how databases work. Sure, you thought this was a trading book but now that I have suckered you into reading a tech book this far, you have no choice but

to bite down and continue.

If you have your own local securities database, you have much better control over your data. It will be so much simpler to handle the data, to fix issues, to add data sources etc.

Luckily, there is a perfectly functional and highly capable database server available free of charge. If you have too much money burning a hole in your pocket, go ahead and select one of the commercial alternatives, but for our purposes you won't see any benefits with that approach. What we are going to use here is **MySQL** , which is not only free but is also available for a wide range of operating systems.

Installing MySQL Server

The examples in this book will use the free **MySQL Community Edition** . If you prefer another database, almost everything should still work more or less

the same way. If you are just starting out and learning about databases, it would be recommended to stick to the **MySQL Community Edition** for now. You can download it though the following URL, which they hopefully haven't changed by the time you read this.

https://dev.mysql.com/downloads/mysql/

Installing is quite straight forward, using a standard visual installation program. Just make sure that the **Python Connectors** are installed, which they should be by default, and don't forget the root password that you will need to set. In my examples here, I will keep that password as simply **root** .

After the installation of the software is done, we can start setting things up. There is a program installed with the **MySQL** server installation called **MySQL Workbench** , which we can use to set up databases and tables.

Start up **MySQL Workbench** and open a connection to your database server. Assuming you installed the server

on the same computer that you are working on, you can connect to hostname **127.0.0.1** , the standard loopback to **localhost** . Enter the username and password that you selected in the installation, and you are all set.

Once you have a connection open to your new database server, you need to create a new database on the server. A **MySQL Server** can run multiple databases, and if you are using the server for many things, it's generally a good idea to keep unrelated tasks separate. In our case, we are setting up a securities database which will contain time-series and metadata for financial markets.

Databases on a database server are often called schemas, and that's what the **MySQL Workbench** likes to call them as well. Create a new database, or schema if you will, and name it something clever. If you haven't spent too much time on the tech side of things, it may surprise you just how much effort techies tend to exert on naming and how clever they feel when they figure out a good naming scheme.

My database, as you see in Figure 24-1, is called mimisbrunn r . This is of course an obvious reference to the Well of Wisdom where Odin sacrificed his…. Oh, fine. Nobody cares. Moving on.

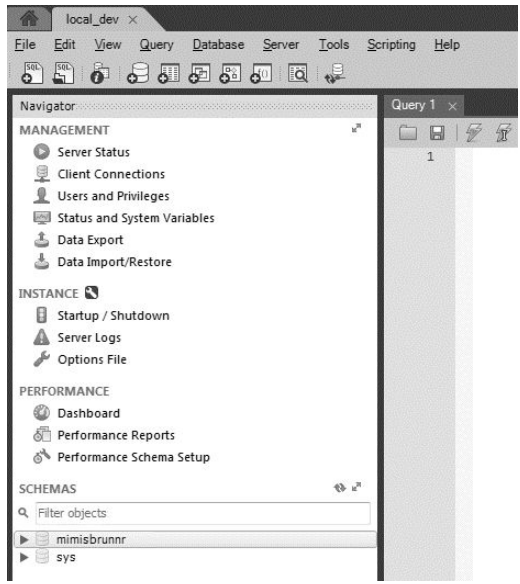


Figure 24-1 MySQL Workbench

Making an Equities Time-Series Table

Now we have a database, but what we are missing still are tables. As you get more comfortable with working with

databases, you will realize that it can be very helpful to have a few different tables for different things. The most straight forward example would be to have an equities time-series table for historical data and a metadata for info such as company name, currency, sector, industry and various code schemes.

For now however, all we need is a simple time-series table for equity type instruments. Create a new table, and call it equity_histor y . In our example, this table will hold the usual time-series fields such as open, high, low, close and volume, but also a couple of other important fields.

I will also add a field for dividends and a field for index membership. The dividend field will be zero most days, and only hold a value if the stock went ex-div that day. As mentioned previously, dividends are very important to take into account for proper backtesting. Hopefully you have already secured a reliable data source which has this information.

The second somewhat different field that I will include in this table is in_sp50 0 .

This field will hold a value of 0 if the stock was not a member of the S&P 500 Index a particular day, or the value 1 if it was part of the index. This way, we can later in our backtests limit our trading to stock which were part of the index on any given day.

Those readers who are already quite familiar with databases are now yelling at the pages that this is not exactly a great way to store such data. There are likely readers who at this point are fuming over how the dividend and index membership data is stored here. And yes, they are of course correct. This is not the best way to do it. But it is the simplest.

I will keep this format for now, as it makes it easier to implement and to explain how things work. It will allow us to move forward quicker and to get the focus back to actual trading.

If you are comfortable enough with databases, feel free to make separate lookup tables for dividend info and index membership. Doing so will make it easier to scale this logic, expanding to cover

more indexes and it will make storage more efficient. But it really does not matter that much for the purposes of this book.

Figure 24-2 Equity History Table

As Figure 24-2 shows, I have set the first two fields, trade_dat e and ticke r , as primary key and unique index. This will both help speed up queries and prevent duplicate entries. As this table is meant for daily data, there should only be one possible row for each stock and day.

Go ahead and create the table. Now we can populate it with some data.

Populating the Database

Luckily, **Pandas** makes talking to our database a breeze. Reading and writing data is very simple using the **Pandas** library. To establish a connection to the database through **Pandas** , a helper library called **SqlAlchemy** can be helpful, so let's start by installing that.

The installation of this library is done using the same methods as we used earlier in this book to install Python libraries. You can open a terminal from the **Anaconda Navigator** , and install from there. Make sure you select the zip3 5 environment, so that we can use this library with our Zipline code.

```
Conda install sqlalchemy
```

Having this library installed, we are now ready to talk to our new database through Python. Depending on what data provider you are dealing with and what kind of delivery option you may have chosen with them, your data may be in a

number of different formats. As it turns out, **Pandas** is quite competent in dealing with this.

As an example, we are going to look at how to turn a turn a flat comma separated file into a neat database table. From this example, you should be able to adapt the code to deal with whatever the format may be of the data that you receive from your provider.

This kind of task is something you will probably find yourself in need of quite often. It's important to understand what is going on here, and how this works, so we'll take it step by step.

We are going to make a function which reads a CSV file from disk and inserts that data into a **MySQL** database. Then we are going to loop through CSV files in a folder and process them one by one.

As a first step, we will read a CSV file and check what it contains. If you have no ready CSV files to use, and you want to try this out, you can use the randomly generated data which can be

downloaded from the book website, **www.followingthetrend.com/trading-evolved** .

This example assumes that you have downloaded the random data from me. Also don't forget to import pandas as p d , just like we have been doing before.

While learning, it can be helpful to write part of the code and then stop to output results. That way, you can make sure that the output is what you expect and you can identify potential issues early on. Try the code below first. Make sure that you update the data_pat h variable to point to where your data is. You can always download my set or random data if you don't have any other data handy.

```
import pandas as pd

data_path = '../data/random_stocks'

def import_file(symbol):
    path = '{}/{}.csv'.format(data_path,symbol)
    df = pd.read_csv(path, index_col=[0],
parse_dates=True)
    return df

df = import_file('A')
df.head()
```

If all worked well, and you had a file called A.cs v in the path specified in the code, you should see something like this.

```
                                    tr
0 1999-01-05 00:00:00 1.0
1 1999-01-06 00:00:00 1.19
2 1999-01-07 00:00:00 1.22
3 1999-01-08 00:00:00 1.23
4 1999-01-11 00:00:00 1.0
```

Look at what this code does. We call the function import_fil e and provide the strin g 'A ' as symbol. The function constructs a path string, pointing to where the file should be.

Pandas then reads the CSV file, and we tell it that the first column, starting with number 0, is the index as well as a data column. Telling **Pandas** to parse the date for us means that we can provide practically any sort of date format, and it will figure out what is day, month and year for us. Finally, we return the finished **DataFrame** .

The final row of the code prints the first ten rows of the **DataFrame** . That's a

useful way to check what is inside, and make sure all worked well. If you prefer to print the last ten rows, you could call .tail( ) instead of .head( ) .

So far so good. Next, we are going to add the code for connecting to the database and writing the data to it.

From a flow point of view, what we want to do here is the following.

- Check which data files are available on disk.
- Read one file at a time, using Pandas.
- Construct an SQL insert statement with the data read.
- Send that insert statement to the server.

For this task, we need to import only three libraries. We need the **OS** library to list the files available, **Pandas** to read the data and **SqlAlchemy** to speak to the database. On top of this, I've imported **tqdm_notebook** , which merely provides a visual progress bar for the Jupiter Notebook environment while we wait.

```
import os
import pandas as pd
from sqlalchemy import create_engine
from tqdm import tqdm_notebook
```

```
engine = create_engine('mysql+mysqlconnector://
root:root@localhost/mimisbrunnr')

data_location = '../data/random_stocks/'
```

Next we'll do a single function which takes a stock symbol as input, reads the data from disk, builds an SQL statement and fires it off. Yes, almost all the logic that we need, in one function.

With this function, I want to show you just how helpful Python can be in making such seemingly complex operations very simple. What I'm going to do here is to construct on giant insert statement, rather than making thousands of small ones. We could fire off one insert statement per day and symbol, but that would be painfully slow. Instead, we'll send one insert statement per stock symbol.

The syntax that we want for this insert statement is shown below. In the example statement below you see three sample days, but we can add a large amount of days this way. As you see, first we define the row header layout, and then

the values for each day.

The final part of the statement deals with what happens in case of duplicates. I added this, as you likely need it. If you try to import data for a stock on a day were your database already has data, you would get an error. With this added instruction at the end of the insert statement, we specify that in case of duplicates, we just overwrite with the latest data.

```
insert into equity_history
(trade_date, ticker, open, high, low, close, volume,
dividend, in_sp500)
values
('2000-01-01', 'ABC', 10, 11, 9, 10, 1000, 0, 1),
('2000-01-02', 'ABC', 10, 11, 9, 10, 1000, 0, 1),
('2000-01-02', 'ABC', 10, 11, 9, 10, 1000, 0, 1),
on duplicate key update
open = values(open),
high = values(high),
low = values(low),
close = values(close),
volume = values(volume),
dividend = values(dividend),
in_sp500 = values(in_sp500);
```

Armed with this knowledge, let's have a look at how our import function builds this potentially giant text string.

The first part, reading the data, should be very familiar by now. After that, we add the initial part of the insert statement.

```
    # First part of the instert statement
    insert_init = """insert into equity_history
        (trade_date, ticker, open, high, low, close,
volume, dividend, in_sp500)
        values
        """
```

So far, nothing out of the ordinary, but the next line of code is where things get clever. This is where we create a giant text string of comma separated values, with each day enclosed in a parenthesis. This row has a lot do digest.

The row starts off with a single character text string, followed by a `join` function. The logic of join is that you set a delimiter, supply a list, and get a delimited string back. For example, `"-".join(['a','b','c'])` will return `a-b-c`.

After that is where we create the list, which you can see by the use of the straight brackets. We iterate each row, using `df.iterrows()`, which will give us the row index and row values for each day. We then put together a text string for each

day, where the .forma t function will insert each value at its designated curly bracket position.

This way of working with Python is very common and if you're struggling with the logic of this, I'd advise you to take a moment and study this next row. Try it out for yourself, make some changes and see what happens.

```
    # Add values for all days to the insert statement
    vals = ",".join(["""('{}', '{}', {}, {}, {}, {}, {}, {},
{})""".format(
        str(day),
        symbol,
        row.open,
        row.high,
        row.low,
        row.close,
        row.volume,
        row.dividend,
        row.in_sp500
    ) for day, row in df.iterrows()])
```

As we now have the bulk of the logic done, all we need to do is to add the final part of the insert statement. That is, the part which updates duplicate values, and finally we put the three pieces of the insert statement together.

```
    # Handle duplicates - Avoiding errors if you've
```

```
already got some data
    # in your table
    insert_end = """ on duplicate key update
        open=values(open),
        high=values(high),
        low=values(low),
        close=values(close),
        volume=values(volume),
        dividend=values(dividend),
        in_sp500=values(in_sp500);"""

    # Put the parts together
    query = insert_init + vals + insert
```

That's it. Then we're ready to send this statement off to the server.

```
    # Fire insert statement
    engine.execute(query)
```

That's all the code we need to convert a csv file with stock data into database rows. But that was just for a single stock, so clearly we need to call this function more than once.

What we'll do is to use the OS library to list the files in the given folder, and then loop through file by file. As mentioned, we'll use the **tqdm** library to create a visual progress bar.

```
"""
Function: get_symbols
```

```
Purpose: Returns names of files in data directory.
"""
def process_symbols():
    # Remember slicing? Let's slice away the last four
    # characters, which will be '.csv'
    # Using [] to make a list of all the symbols
    symbols = [s[:-4] for s in os.listdir(data_location)]
    for symbol in tqdm_notebook(symbols,
desc='Importing stocks...'):
        import_file(symbol)
```

```
process_symbols()
```

That's all we need. As usual, I will show the entire code at once below.

```
import os
import pandas as pd
from sqlalchemy import create_engine
from tqdm import tqdm_notebook

engine = create_engine('mysql+mysqlconnector://
root:root@localhost/mimisbrunnr')

data_location = '../data/random_stocks/'

"""
Function: import_file
Purpose: Reads a CSV file and stores the data in a
database.
"""
def import_file(symbol):
    path = data_location + '{}.csv'.format(symbol)
    df = pd.read_csv(path, index_col=[0],
```

```
parse_dates=[0])
```

```
    # First part of the instert statement
    insert_init = """insert into equity_history
        (trade_date, ticker, open, high, low, close,
volume, dividend, in_sp500)
        values
        """
```

```
    # Add values for all days to the insert statement
    vals = ",".join(["""('{}', '{}', {}, {}, {}, {}, {}, {},
{})""".format(
        str(day),
        symbol,
        row.open,
        row.high,
        row.low,
        row.close,
        row.volume,
        row.dividend,
        row.in_sp500
    ) for day, row in df.iterrows()])
```

```
    # Handle duplicates - Avoiding errors if you've
already got some data
    # in your table
    insert_end = """  on duplicate key update
        open=values(open),
        high=values(high),
        low=values(low),
        close=values(close),
        volume=values(volume),
        dividend=values(dividend),
```

```
        in_sp500=values(in_sp500);"""

    # Put the parts together
    query = insert_init + vals + insert
```

```
    # Fire insert statement
    engine.execute(query)
    """
Function: get_symbols
Purpose: Returns names of files in data directory.
"""
def process_symbols():
    # Remember slicing? Let's slice away the last four
    # characters, which will be '.csv'
    # Using [] to make a list of all the symbols
    symbols = [s[:-4] for s in os.listdir(data_location)]
    for symbol in tqdm_notebook(symbols,
desc='Importing...'):
        import_file(symbol)
```

```
process_symbols()
```

Querying the Database

Now that we have a nice time-series database, we can easily and quickly access it. While we do need to use SQL to speak to our database, we really have no need

here for any deeper SQL skills. A very basic understanding of this query language will suffice.

To demonstrate, we are going to write some code to fetch time-series from the database and display a chart. Easy stuff.

The basic syntax for asking the database for information is something like this.

```
SELECT fields FROM table WHERE conditions
```

Of course, SQL can do a whole lot more if you want it to. One reason why I recommend using a proper database instead of dealing directly with flat files is that you have so many more possibilities. As you dig deeper and expand your knowledge, you will find that using a MySQL database will greatly help when you start building more complex models or expand your data coverage.

The following code will fetch the time-series history for ticker AAPL and chart it, using the same techniques as we learned earlier in this book. Note that I threw in an example of a new way to slice

a **Pandas DataFrame** as well. In this code, on the second last line, I select all data between 2014 and 2015 and single that out for charting.

```
%matplotlib inline
import pandas as pd
from  matplotlib import pyplot as plt, rc
from sqlalchemy import create_engine

engine = create_engine('mysql+mysqlconnector://
root:root@localhost/mimisbrunnr')

# Chart formatting, used for book image
font = {'family' : 'eurostile',
        'weight' : 'normal',
        'size'   : 16}
rc('font', **font)

# Function to fetch history from db
def history(symbol):
    query = """select trade_date, close, volume
        from equity_history where ticker='{}'
    """.format(symbol)
    print('This is the SQL statement we send: \n
{}'.format(query))
    df = pd.read_sql_query(query, engine,
index_col='trade_date', parse_dates=['trade_date'])
    return df
```

```
# Fetch data for the stock
ticker = 'AAPL'
hist = history(ticker)
```

```
# Set up the chart
fig = plt.figure(figsize=(15, 8))
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_title('Chart for {}'.format(ticker))

# Note how you can use date ranges to slice the time-
series
plot_data = hist['2014-01-01':'2015-01-01']

# Plot the close data
ax.plot(plot_data.close, linestyle='-', label=ticker,
linewidth=3.0, color='black')
ax.set_ylabel("Price")

# Make a second Y axis, sharing the same X
ax2 = ax.twinx()
ax2.set_ylabel("Volume")

# Plot volume as bars
ax2.bar(plot_data.index, plot_data.volume,
color='grey')
```

This code will output the SQL query itself, just to show you clearly how the finished query string looks.

This is the SQL statement we send:

```
select trade_date, close, volume
    from equity_history where ticker='AAPL'
```

Finally it will output a now familiar looking graph. I added a second y-axis

with volume information as well, just to show how that can be done.

If you wonder why the AAPL price looks a little different than you may remember, there is a good reason for this. In this example, I used the random data from my website, to allow readers to start working and learning even if they don't yet have a proper data source. This is randomly generated data, using a **Random Walk** methodology.

Figure 24-3 Random Apples

As you see here, simply asking the database for time-series history of a given ticker is very straightforward. At the moment, that's really all the SQL that you need to know. There is a clear benefit of understanding a bit more SQL, but with

this chapter I merely want to get you interested in the subject.

Making a Database Bundle

The logic for importing data from a securities database into Zipline should look quite familiar by now. The code here below is practically the same as when we read data from csv files earlier. The only real difference is that we are now reading data from the database, instead of flat files.

When we read csv files from disk, we used the function pd.read_csv( ) and the main differences here is that we instead use pd.read_sql_query( ) to fetch data. We will provide this function with a database connection and a simple SQL query to specify what data we want to fetch.

Given that we populated our brand new database with stocks earlier in this chapter, we can now just ask the database which exact stocks are available. The

function below shows how that can be done with minimum code.

```
from sqlalchemy import create_engine

engine = create_engine('mysql+mysqlconnector://
root:root@localhost/mimisbrunnr')

def available_stocks():
    symbol_query = "select distinct ticker from
equity_history order by ticker"
    symbols = pd.read_sql_query(symbol_query, engine)
    return symbols.ticker # Returns a list of tickers
```

The method of fetching a list of available stocks is one of two parts where the code differs from the csv equity bundle we saw earlier. The second part that differs is how we read data for each stock. Where we previously read one file per stock, we now fire off one query to the database per stock.

```
        # Make a database query
        query = """select
                    trade_date as date, open, high, low, close,
volume, dividend
                    from equity_history where ticker='{}'
order by trade_date;
            """.format(symbol)
```

```
        # Ask the database for the data
        df = pd.read_sql_query(query, engine,
```

```
index_col='date', parse_dates=['date'])
```

That's really the only thing that differs. Even though it's materially the same, I include the full source code for the database bundle here below. Naturally, you could do the same thing with the futures data as well.

```
import pandas as pd
from tqdm import tqdm # Used for progress bar
from sqlalchemy import create_engine

engine = create_engine('mysql+mysqlconnector://
root:root@localhost/mimisbrunnr')

def available_stocks():
    symbol_query = "select distinct ticker from
equity_history order by ticker"
    symbols = pd.read_sql_query(symbol_query, engine)
    return symbols.ticker # Returns a list of tickers
```

```
"""
The ingest function needs to have this exact signature,
meaning these arguments passed, as shown below.
"""
def database_bundle(environ,
                    asset_db_writer,
                    minute_bar_writer,
                    daily_bar_writer,
                    adjustment_writer,
                    calendar,
```

```
            start_session,
            end_session,
            cache,
            show_progress,
            output_dir):
```

```
# Get list of files from path
# Slicing off the last part
# 'example.csv'[:-4] = 'example'
symbols = available_stocks()
```

```
# Prepare an empty DataFrame for dividends
divs = pd.DataFrame(columns=['sid',
                             'amount',
                             'ex_date',
                             'record_date',
                             'declared_date',
                             'pay_date']
)
```

```
# Prepare an empty DataFrame for splits
splits = pd.DataFrame(columns=['sid',
                               'ratio',
                               'effective_date']
)
```

```
# Prepare an empty DataFrame for metadata
metadata = pd.DataFrame(columns=('start_date',
                                 'end_date',
                                 'auto_close_date',
                                 'symbol',
                                 'exchange'
```

```
                                )
                            )


    # Check valid trading dates, according to the
selected exchange calendar
    sessions = calendar.sessions_in_range(start_session,
end_session)
```

```
    # Get data for all stocks and write to Zipline
    daily_bar_writer.write(
            process_stocks(symbols, sessions, metadata,
divs)
            )

    # Write the metadata
    asset_db_writer.write(equities=metadata)
```

```
    # Write splits and dividends
    adjustment_writer.write(splits=splits,
                            dividends=divs)
```

```

"""
Generator function to iterate stocks,
build historical data, metadata
and dividend data
"""
def process_stocks(symbols, sessions, metadata, divs):
    # Loop the stocks, setting a unique Security ID (SID)
```

```
    sid = 0
```

```
    for symbol in tqdm(symbols):
        sid += 1
```

```
        # Make a database query
        query = """select
                    trade_date as date, open, high, low, close,
volume, dividend
                    from equity_history where ticker='{}'
order by trade_date;
              """.format(symbol)
```

```
        # Ask the database for the data
        df = pd.read_sql_query(query, engine,
index_col='date', parse_dates=['date'])

        # Check first and last date.
        start_date = df.index[0]
        end_date = df.index[-1]
```

```
        # Synch to the official exchange calendar
        df = df.reindex(sessions.tz_localize(None))
[start_date:end_date]
```

```
        # Forward fill missing data
        df.fillna(method='ffill', inplace=True)
```

```
        # Drop remaining NaN
        df.dropna(inplace=True)
```

```
        # The auto_close date is the day after the last
```

```
trade.
        ac_date = end_date + pd.Timedelta(days=1)

        # Add a row to the metadata DataFrame.
        metadata.loc[sid] = start_date, end_date, ac_date,
symbol, 'NYSE'

        # If there's dividend data, add that to the
dividend DataFrame
        if 'dividend' in df.columns:

            # Slice off the days with dividends
            tmp = df[df['dividend'] != 0.0]['dividend']
            div = pd.DataFrame(data=tmp.index.tolist(),
columns=['ex_date'])

            # Provide empty columns as we don't have this
data for now
            div['record_date'] = pd.NaT
            div['declared_date'] = pd.NaT
            div['pay_date'] = pd.NaT

            # Store the dividends and set the Security ID
            div['amount'] = tmp.tolist()
            div['sid'] = sid

            # Start numbering at where we left off last
time
            ind = pd.Index(range(divs.shape[0],
divs.shape[0] + div.shape[0]))
            div.set_index(ind, inplace=True)
```

```
            # Append this stock's dividends to the list of
all dividends
            divs = divs.append(div)

        yield sid, df
```

25 Final Words – Path Forward

This has been quite a long book, and hopefully one packed with new information for most readers. If you read the book all the way through here, and struggle to absorb all of that contents, there's no need to worry. For most readers, this book contains a substantial amount of new information, and I would expect that they would need to read the book multiple times, and try the sample code before gaining a full understanding. It's a lot to take in at once.

As mentioned in the introduction to this book, there is no substitute for actual, practical experience. To get full use of this

book, I strongly recommend that you actually set up an environment as described in this book. Install the software, import your data and replicate the models.

Perhaps your interest is not at all in the kind of trading models presented in this book. That's perfectly fine, and it will probably be the case for many readers. But if you start by replicating these models, you will have a solid foundation to build upon, when you move on to code your own ideas.

A book of this nature cannot possibly teach you everything, either about trading or about Python. It can only scratch the surface of both, and hopefully peak your interest in moving further on your own.

Build Your Own Models

You probably have some interesting trading ideas. Perhaps you have

constructed models before in one of the usual suspects in the line-up of retail trading software that you want to try out in Python. Translating an existing model from a different platform is a great exercise.

If you already have such a model, start off by translating it to Python. That way, you can easily compare step by step, to make sure you get the output and results that you expect. When learning, it makes debugging much easier if you already know what results you're looking for.

If you don't yet have any models created on other platforms, you probably still have trading ideas. It can be quite an eye opener the first time that you try to formulate precise rules for your trading ideas, and it can be a very valuable experience.

What happens for most people is that it forces you to define the rules in much more detail than you had expected. You might have been trading these rules for years without realizing that there was

some discretionary component to it. Once you boil it down to exact math, you will without a doubt find out if this is the case, and that can be a great step towards becoming a full-fledged systematic trader.

Other Backtesting Engines

The focus on this entire book has been on a single backtesting engine, and I suspect that will earn me a few one star Amazon reviews. It would be nice to be able to provide instructions and sample code for many different backtesting software packages, but that would simply not be feasible in a book like this. It would either make the book five times as long, or reduce actual modeling a brief overview.

I did consider to cover just two backtesting engines, as there are at this time two of them competing in the race for most advanced backtester. The other engine in that case would have been

LEAN, built and maintained by QuantConnect.

LEAN is an open source backtesting engine, much like Zipline, and it too has an online version where you get free access to data as well. Just as you can use the Quantopian website to run backtests in their hosted environment with their data, you can do the same on the QuantConnect website.

From a technical point of view however, the two engines are extremely different. While Zipline is a native Python solution, LEAN is developed in C# and lets you choose between multiple programming languages for writing algos.

The process of installing and configuring LEAN is very different from Zipline and out of scope for this particular book. This is a solid backtesting engine though, and I hope to write a bit more about it on my website.

Clearly the ability to write algos in multiple languages is a big plus for QuantConnect, but the flipside of that is that you're missing the native Python

experience.

If you would like to try out LEAN without investing too much time up front, you could go to the QuantConnect website (QuantConnect, 2019) and try out some sample code on their data.

Another widely used backtesting engine is Backtrader (Backtrader, 2019), which also is open source and completely free. There is an active Backtrader community and you should be able to find sample code and help in online forums.

But do you need to try out other backtesting engines? Well, that all depends on your own trading styles, ideas and requirements. For many systematic traders, Zipline can do anything you ever want to do. Other traders may have already hit a deal breaker in the introductory chapters.

Perhaps you're only interested in options strategies, or spot forex. Perhaps you need multi-currency asset support, or perhaps there are other reasons for you to pursue a different backtesting engine.

In my own view, no backtesting engine is perfect. There is no one stop shop. While I find Zipline to be a great piece of software, this may or may not be true for you and your situation.

After reading this book, you should have enough grounding in Python and backtesting, that you should be able to figure out if you need something different, and how to go about setting up another backtester.

How to Make Money in the Markets

In closing of this book, I'd like to reiterate a point that I try to make when speaking at conferences around the world. Most people who want to make money from the financial markets have misunderstood where the real money is. And it's not in trading your own account.

Extremely few people have ever become financially independent by trading their own account. You become

financially independent by trading other people's money.

The gist of this argument is that if you trade your own money, you have a limited upside and take all the risk. A highly skilled professional trader is likely to see returns in the range of 12-18% per year over time, with occasional drawdowns of about three times that. Expecting triple digits returns with low risk is not grounded in reality.

However, if you trade other people's money, perhaps along with your own, you have a near unlimited upside and a limited downside. You can scale up by managing more money, multiplying your income potential. In comparison, the decision to manage only your own money amounts to a poor trading decision.

I don't merely bring this up to provoke you to think in a different direction. This is a genuine advice. Trading other people's assets for a living is the way to go, from a financial perspective. That's where the interesting money in trading comes from.

The types of models described in this book are purposely selected with this in mind. Futures models for instance may be difficult to implement with a small private account, but they can be quite interesting for professional asset management.

If you are early in your career and just starting out with all of this financial modeling and backtesting, my number one advice to you is to consider going the professional route, and aim to manage other people's money for a living.

References

Anaconda Download . (n.d.). Retrieved from https://www.anaconda.com/download/

Backtrader . (2019, June). Retrieved from https://www.backtrader.com/

Carver, R. (2015). *Systematic Trading.* Harriman House.

Carver, R. (2019). *Leveraged Trading.*

Harriman House.
Clenow, A. (2013). *Following the Trend.* Wiley.
Clenow, A. (2015). *Stocks on the Move.*
Grinold, R. (1999). *Active Portfolio Management.* Wiley.
Hilpisch, Y. (2018). *Python for Finance.* O'Reilly Media.
Lewis, M. (1989). *Liar's Poker.*
McKinney, W. (2017). *Python for Data Analysis.*
QuantConnect . (2019, June). Retrieved from https://quantconnect.com
SPIVA . (2019). Retrieved from https://us.spindices.com/spiva/

Software Versions Used

This section is provided for those who seek to replicate the exact environment that I used for this book.

That shouldn't be necessary for most of you, but I provide it here just in case.

The book models were written on a development machine with Windows 10 installed. It was also tested on another machine with Windows 7 and some different configurations. Below are the versions of Python packages used for the zip35 environment in the book.

Python version 3.5 and conda version 4.5.10 was used throughout.

```
# packages in environment at C:\ProgramData
\Anaconda3_new\envs\zip35:
#
# Name                    Version                   Build
Channel
_nb_ext_conf              0.4.0                     py35_1
alabaster                 0.7.10                    py35_0
alembic                   0.7.7                     py35_0
Quantopian
alphalens                 0.3.6                     py_0
conda-forge
anaconda-client           1.6.14                    py35_0
asn1crypto                0.24.0                    py35_0
astroid                   1.5.3                     py35_0
babel                     2.5.0                     py35_0
backcall                  0.1.0                     py35_0
bcolz                     0.12.1                    np111py35_0
```

Quantopian
blas 1.0 mkl
bleach 2.1.3 py35_0
blosc 1.14.3 he51fdeb_0
bottleneck 1.2.1
py35h452e1ab_1
bzip2 1.0.6 hfa6e2cd_5
ca-certificates 2019.3.9
hecc5488_0 conda-forge
certifi 2018.8.24 py35_1001
conda-forge
cffi 1.11.5 py35h74b6da3_1
chardet 3.0.4 py35_0
click 6.7 py35h10df73f_0
clyent 1.2.2 py35h3cd9751_1
colorama 0.3.9 py35h32a752f_0
contextlib2 0.5.5
py35h0a97e54_0
cryptography 2.3.1
py35h74b6da3_0
cycler 0.10.0 py35_0
cyordereddict 0.2.2 py35_0
Quantopian
cython 0.28.3 py35hfa6e2cd_0
decorator 4.3.0 py35_0
docutils 0.14 py35_0
empyrical 0.5.0 py35_0
quantopian
entrypoints 0.2.3
py35hb91ced9_2

freetype 2.9.1 ha9979f8_1
funcsigs 0.4 py35_0
hdf5 1.10.2 hac2f561_1
html5lib 1.0.1 py35h047fa9f_0
icc_rt 2017.0.4 h97af966_0
icu 58.2 ha66f8fd_1
idna 2.7 py35_0
imagesize 0.7.1 py35_0
intel-openmp 2018.0.3 0
intervaltree 2.1.0 py35_0
Quantopian
ipykernel 4.8.2 py35_0
ipython 6.4.0 py35_0
ipython_genutils 0.2.0
py35ha709e79_0
ipywidgets 7.2.1 py35_0
isort 4.2.15 py35_0
jedi 0.12.0 py35_1
jinja2 2.10 py35hdf652bb_0
jpeg 9b hb83a4c4_2
jsonschema 2.6.0
py35h27d56d3_0
jupyter_client 5.2.3 py35_0
jupyter_core 4.4.0
py35h629ba7f_0
kiwisolver 1.0.1 py35hc605aed_0
lazy-object-proxy 1.3.1 py35_0
libiconv 1.15 h1df5818_7
libpng 1.6.37 h7602738_0
conda-forge

libsodium 1.0.16 h9d3ae62_0
libxml2 2.9.8 hadb2253_1
libxslt 1.1.32 hf6f1972_0
logbook 0.12.5 py35_0
Quantopian
lru-dict 1.1.4 py35_0
Quantopian
lxml 4.2.5 py35hef2cd61_0
lzo 2.10 h6df0209_2
m2w64-gcc-libgfortran 5.3.0 6
m2w64-gcc-libs 5.3.0 7
m2w64-gcc-libs-core 5.3.0 7
m2w64-gmp 6.1.0 2
m2w64-libwinpthread-git
5.0.0.4634.697f757 2
mako 1.0.7 py35ha146b58_0
markupsafe 1.0
py35hc253e08_1
matplotlib 2.2.2 py35had4c4a9_2
mistune 0.8.3 py35hfa6e2cd_1
mkl 2018.0.3 1
mock 2.0.0 py35h0f49239_0
msys2-conda-epoch 20160418
1
multipledispatch 0.5.0 py35_0
mysql-connector-python 2.0.4
py35_0
nb_anacondacloud 1.4.0 py35_0
nb_conda 2.2.0 py35_0
nb_conda_kernels 2.1.0 py35_0

nbconvert 5.3.1
py35h98d6c46_0
nbformat 4.4.0
py35h908c9d9_0
nbpresent 3.0.2 py35_0
networkx 1.11
py35h097edc8_0
norgatedata 0.1.45
py35h39e3cac_0 norgatedata
nose-parameterized 0.6.0
<pip>
notebook 5.5.0 py35_0
numexpr 2.6.8 py35h9ef55f4_0
numpy 1.11.3
py35h53ece5f_10
numpy-base 1.11.3
py35h8128ebf_10
numpydoc 0.7.0 py35_0
openssl 1.0.2s he774522_0
pandas 0.22.0
py35h6538335_0
pandas-datareader 0.6.0 py35_0
pandoc 1.19.2.1 hb2460c7_1
pandocfilters 1.4.2
py35h978f723_1
parso 0.2.1 py35_0
patsy 0.5.0 py35_0
pbr 4.0.4 py35_0
pickleshare 0.7.4 py35h2f9f535_0
pip 19.2.3 <pip>

pip 9.0.1 py35_1
plotly 2.0.11 py35_0
prompt_toolkit 1.0.15
py35h89c7cb4_0
psutil 5.2.2 py35_0
pycodestyle 2.3.1 py35_0
pycparser 2.19 py35_0
pyflakes 1.6.0 py35_0
pyfolio 0.9.0 <pip>
pygments 2.2.0
py35h24c0941_0
pylint 1.7.2 py35_0
pyopenssl 18.0.0 py35_0
pyparsing 2.2.0 py35_0
pyqt 5.9.2 py35h6538335_2
pysocks 1.6.8 py35_0
pytables 3.4.4 py35he6f6034_0
python 3.5.5 h0c2934d_2
python-dateutil 2.7.3 py35_0
pytz 2018.4 py35_0
pywinpty 0.5.4 py35_0
pyyaml 3.12 py35h4bf9689_1
pyzmq 17.0.0 py35hfa6e2cd_1
qt 5.9.6 vc14h1e9a669_2
qtawesome 0.4.4 py35_0
qtconsole 4.3.1 py35_0
qtpy 1.3.1 py35_0
requests 2.14.2 py35_0
requests-file 1.4.3 py35_0
requests-ftp 0.3.1 py35_0

rope 0.9.4 py35_1
scikit-learn 0.19.1
py35h2037775_0
scipy 1.1.0 py35hc28095f_0
seaborn 0.8.1 py35hc73483e_0
send2trash 1.5.0 py35_0
setuptools 36.4.0 py35_1
simplegeneric 0.8.1 py35_2
singledispatch 3.4.0.3 py35_0
sip 4.19.8
py35h6538335_1000 conda-forge
six 1.11.0 py35hc1da2df_1
snappy 1.1.7 h777316e_3
snowballstemmer 1.2.1 py35_0
sortedcontainers 1.4.4 py35_0
Quantopian
sphinx 1.6.3 py35_0
sphinxcontrib 1.0 py35_0
sphinxcontrib-websupport 1.0.1
py35_0
spyder 3.2.3 py35_0
sqlalchemy 1.2.8
py35hfa6e2cd_0
sqlite 3.28.0 hfa6e2cd_0
conda-forge
statsmodels 0.9.0
py35h452e1ab_0
terminado 0.8.1 py35_1
testfixtures 6.2.0 <pip>
testpath 0.3.1 py35h06cf69e_0

tk 8.6.7 hcb92d03_3
toolz 0.9.0 py35_0
tornado 5.0.2 py35_0
tqdm 4.26.0 py35h28b3542_0
trading-calendars 1.0.1 py35_0
quantopian
traitlets 4.3.2 py35h09b975b_0
urllib3 1.23 py35_0
vc 14.1 h0510ff6_3
vs2015_runtime 15.5.2 3
wcwidth 0.1.7
py35h6e80d8a_0
webencodings 0.5.1
py35h5d527fb_1
wheel 0.29.0 py35_0
widgetsnbextension 3.2.1 py35_0
win_inet_pton 1.0.1 py35_1
win_unicode_console 0.5
py35h56988b5_0
wincertstore 0.2 py35_0
winpty 0.4.3 4
wrapt 1.10.11 py35_0
yaml 0.1.7 hc54c509_2
zeromq 4.2.5 hc6251cf_0
zipline 1.3.0 np111py35_0
quantopian
zlib 1.2.11 h8395fce_2

Index

Allocation, 20
Anaconda, 39
Anaconda package, 54
analyze, 106
Analyzing Backtest Results, 117
Annualized carry, 309
Annualized return, 137
arange, 185
asset allocation model, 161
Asset Class, 18
Average True Range, 187
Backtesting, 84
Backtesting platform, 84
Backwardation, 301
Bribing the Author, 88
Bundles, 372
CAGR, 137

Code, Block of, 44
Coefficient of determination, 179
Combining Models, 320
Command Line Installation, 94
Compound Annual Growth Rate, 137
Conditional Logic, 50
contango, 301
Contract size, 215
Copy to Clipboard, 80
Correlation, 71
Creating New Environment, 93
CSI Data, 85
Cumulative Product, 67
Currency exposure, 219
Custom Analysis, 130
Data, 17, 371
Data Used, 116
Databases, 395
DataFrame, 59
Dictionary, 47

dividends, 170
Documentation, 61
Dynamic Performance Chart, 270
Empyrical, 330
Entry Rules, 20
Environments, 92
Equity Investment Universe, 170
ETF, 143
ETF trading models, 159
Excel, 60
Exchange traded funds, 143
Exchange Traded Notes, 145
Exit Rules, 20
Exponential regression slope, 179
Financial risk. *See* Risk
first trading algorithm, 102
Function Definition, 72
Futures bundle, 382
Futures continuations, 227
gross_leverage, 140

handle_data, 104
Head, 80
history, 104
if statement, 105
IndendationError, 52
Indentation, 45
Index constituents, 171
Index trackers, 143
Ingesting, 98
Installation of Python libraries, 54
Installing Python, 38
Installing Zipline, 91
Interpreted language, 36
Investment Universe, 19
iterrrows(), 113
Jupyter, 46
Jupyter Notebook, 40
Leverage, 220
Linear regression slope, 180
List, 44

loc, 112
Loop, 44
Margin, 217
Mark to Market, 25
MatPlotLib , 56
Maximum drawdown, 138
Momentum, 175
Momentum score, 179
Money Management. *See* Risk Fallacies
Monkeys, 337
Moving Average, 58
Multiplier, 217
MySQL, 398
nb_conda, 100
Norgate Data, 85
Numpy, 65
order_target, 105
order_target_percent, 103
Pandas , 57
Pct_Change, 67

Plot, 68
Plotting, 76
Portfolio Backtest, 109
portfolio_value, 140
Pullback, 288
PyFolio, 117
Pyfolio Tear Sheets, 124
Pyramiding. *See* Risk Fallacies
Quandl Bundle, 98
QuantCon, 88
Quantopian, 88
QuantQuote, 85
range(), 152
read_sql_query, 412
Reading CSV Files, 58
rebalancing, 21
Relative Strength, 80
RightEdge, 85
Risk, 23
Risk Fallacies, 26

Risk per Trade . *See* Risk Fallacies
Rolling, 231
Rolling Correlation, 75
Rolling mean, 191
Rolling Mean, 58
Rolling time window, 138
run_algorithm, 103
scipy, 185
Securities Database, 398
Shift, 67
Short ETFs, 150
Slicing, 74
Snapshot, 132
SQL insert statement, 405
Standard deviation, 187
Standardization, 213
String Concatenation, 49
Support, 6
symbol, 103
Tail, 80

Term Structure, 299
to_clipboard, 329
to_csv, 328
Trend Following, 234
Volatility based allocation, 186
Zipline, 88